Robotic Navigation

M.Sc. Thesis

by

John Ó Duinn
Dublin City University

Academic Supervisor
Mícheál Ó hÉigeartaigh
School of Computer Applications

3.2. Standard Lee's Algorithm 38
3.3. Lee's Algorithm with Straight Line Bias 38
4. Implementations based on Graph Theory
4.1. The Partial Update Technique Explained 40
4.2. Testbed Implementation Details 42
4.2.1. How the Domain was Represented 43
4.2.2. The Camera Image Generating Module 45
4.2.3. How a Graph Network was Generated from a Camera Image 45
4.2.4. How the Graph Network was Represented 46
4.2.5. How the Graph Network was Updated After Every Timeslice 47
4.2.6. Keeping track of nodes to visit 48
4.3. Graph Search Algorithms Used 49
5. Benchmarks and Performance Analysis
5.1. Test Data Used 50
5.2. Test Results Recorded 52
5.3. Analysis of Benchmarks 53
5.3.1. Single AGV in a Static Domain 53
5.3.2. Single AGV in a Domain with Independent Mobile Obstacles 54
5.3.3. Multiple AGVs in the same Domain 56
6. Conclusion 57
7. References 59
A. Benchmarks
A.1. Single AGV in a Static Domain A-2
A.2. Single AGV in a Domain with Independent Mobile Obstacles A-5
B. Source Code Listings
B.1. Standard Lee's Algorithm B-1
B.2. Modified Lee's Algorithm B-22
B.3. Standard Breadth First Graph Theory B-47
B.4. Standard Depth First Graph Theory B-82
B.5. Standard Priority First Graph Theory B-117
B.6. Standard A* Graph Theory B-153
B.7. Partial Update Breadth First Graph Theory B-189
B.8. Partial Update Depth First Graph Theory B-229
B.9. Partial Update Priority First Graph Theory B-269
B.10. Partial Update A* Graph Theory B-310

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science in Computer Applications is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:____________________________

ID: 90700881

Date: ____________________________

An Abstract Of "Robotic Navigation"

by

John Ó Duinn

The aim of this project was to design software which could be used to control autonomous guided vehicles (AGVs) through a domain without colliding with any obstacles which might be present. The domains modelled included both mobile and stationary obstacles. This was in order to more realistically represent environments such as Automated Warehouses, in which multiple, independently controlled, AGVs travelled while avoiding collisions with both stationary obstacles such as pallets and mobile obstacles such as other AGVs and humans.

Several different existing techniques were reviewed in detail for suitability including Lee's algorithm, Generalised Voronai Diagrams and Neural Networks. Those techniques which showed suitability for navigation were implemented. Extensions and modifications were developed to improve the performance of these techniques and implemented. Sample domains were built, and the performance of the textbook algorithms compared against the performance of the newly developed variants. Benchmarks were tabulated and analysed.

Automated warehouses are good examples of the use of more sophisticated AGVs. In these scenarios, the AGV's workspace is the entire warehouse floor. The size and location of all objects in this warehouse are accurately known as they have been placed there by the AGV. If a human enters the workspace, the AGV must be suspended for safety reasons until the human has left the area.

Both these types of robots share some fundamental traits. The environment in which they work must be totally known. All mobile objects in the domain must travel in a pre-defined manner, without any deviations in either direction or velocity. These systems are not capable of dealing with unforeseen or unplanned movements within their environment. During installation, they have to be guided through the set of movements which they are required to follow. Once these movements have been entered, the robot will simply loop through them precisely and endlessly. Some companies have developed a computer software interface so that the movements could be directly programmed, and thus save on the time and expense of manually leading the robot through it's entire set of movements., but the functional capabilities of the robot are essentially the same.

However, it is important to point out that these robots have no ability to react to or interact with the surrounding environment and therefore these robots can only be successfully used where the working environment is totally static. This lack of flexibility is the main reason why robots cannot be used interactively in environments shared with humans.

1.2. Future Robots and their uses

More flexible and adaptable robots are needed which can interact with and react to an existing and changeable environment. This flexibility is essential before robots can be taken from a carefully controlled factory floor and introduced into mainstream human life.

Take for example the AGVs mentioned above. By travelling along pre-planned and controlled routes, with most unknown factors (i.e. humans) carefully excluded from interfering, these AGVs just blindly trundled up and down the same pre-planned routes. If an obstacle was encountered, the AGV simply stopped and signalled an alert (either by siren or radio link to a control centre). Human intervention was then required before the AGV could travel again. This type of AGV had no functional place in our fast changing and unpredictable human society and would be forever limited to carefully controlled factory environments.

By contrast, an AGV which was able to adapt and react to a changing environment could serve a purpose in our society. In order to be useful, it would have to:

(i) Successfully travel in a changeable human-orientated environment with little or no human assistance.

(ii) Recognise and avoid obstacles which it may encounter. These may be static (i.e. a chair which has been moved) or mobile (the family dog).

(iii) Perform useful and mundane tasks which would help it gain acceptance from the general community.

A lot of research is currently underway to achieve these capabilities. One well publicised project in this area is NASA's Mars Exploration robot. The project aims to have an AGV explore the surface of Mars with minimal human guidance. This independence is required because of the time delay in transmitting signals from Mars to Earth and transmitting the control signals back. Another project, CMU's NAVILAB, drives a van down a twisted Campus roadway unaided. These systems indicate some possible future uses for robots in a changeable domain.

1.3. Path Planning versus Path Following

Regardless of the domain in which they will be used, all AGVs needed some way of deciding how to reach the goal. The currently popular "trail following" technique was clearly unsuitable for more realistic and unstructured environments. Therefore, one of the most obvious and important areas in which all AGVs needed improvement was in the area of navigation software. Having an AGV which could plan it's own route to the destination was a big improvement over AGVs which could only follow a pre-marked path.

Path planning involves the examining of a domain to determine what was the best path from the AGV to the goal, depending on current criteria (i.e. shortest, fastest, etc.). Once a path to the goal was known, path following ensured that the AGV adhered to this path - within the physical limitations of the AGV's steering and navigation mechanisms. Readers interested in a general introduction to the field of path planning are refered to [JOR86], [KNU73], [SED90] and [WIN84]. Also, several path planning algorithms are discussed in detail in Chapter 2.

This project concentrates on path planning and not path following. Readers interested in the application of expert systems, fuzzy logic, realistic modelling algorithms, neural networks, and other techniques to the area of path following are referred to [FEN91], [HEM92], [KRA85], [HUN91], [NIE91], [TAK89]. Path following was the basic concept behind the "road following" type projects covered in [THO91] [TOU89] and [WAX87].

The navigation software required to do this should also be able to find a reasonably short path to the goal, and also avoid any mobile obstacles which may intrude en route. This last feature was quite important for the AGV to be able to function in more realistic environments.

1.4. AGV and Manipulator Navigation

The navigation techniques used for navigating an AGV and a manipulator are quite different. This is mainly due to restrictions imposed on the navigation software by the physical movement abilities of a manipulator arm.

This project will be investigating different navigation techniques for AGVs. All references to navigation which occur in the rest of this paper refer to AGV navigation. Readers interested in works on manipulator navigation are referred to [CHE89], [HAW86], [KHE88], [LAU84], [LUM85], [MOR90], [OZA84], [PAU81], [VAN84] and [ZOM92].

1.5. Objectives of this Paper

The original idea behind this paper was to investigate how an AGV could navigate through a unstructured environment. This was basically research into the area of path planning in an environment with both mobile and stationary obstacles. The AGV would try to navigate from it's initial location to a given goal location without colliding with any obstacles or possibly even other AGVs, which may be encountered en route. During the course of this paper, we experimented with three different operating environments or models. They were:

(i) Single AGV with Stationary Obsticles. This model contained only one AGV and several static obstacles. At no stage could were there any mobile obstacles present.

(ii) Single AGV with Mobile Obstacles. This model contained only one AGV. However, it also contained a mix of both static and mobile obstacles. It was the responsibility of the AGV to detect a pending collision with any of these mobile obsticles and take steps to avoid them.

(iii) Multiple AGVs. This model contained multiple AGVs, and several static obstacles. Each AGV had it's own intended destination to reach, and treated any other AGVs encountered en route in exactly the same manner as any other mobile obstacle.

Different criteria could be selected at run-time to decide what made one valid path better then another. These criteria would include the shortest path, the fastest path, the path with the least number of turns and

the path with the greatest safety margin. This was easily implemented by charging a cost to travel from one point to another. The value of this cost depended on what the current criteria was. If minimum distance was the primary concern, then the cost was the distance between the two points. However if safety was the primary concern, then the cost was inversely proportional to the distance of the nearest obsticle. Regardless of the criteria, the path which the AGV travelled should be reasonably close to the cheapest, and ideally would actually be the cheapest path possible to the goal.

The time taken to find an acceptable path to the goal was to be kept as short as possible. Ideally it would be found in real-time, but this was unlikely. Attempts to achieve this included developing a new technique for this application, modifying an existing technique or investigating a solution which was executed in parallel.

Finally, we would like to point that this project was only concerned with the software aspects of path planning. Hardware aspects were not dealt with in this paper at all.

1.6. Assumptions made in this Paper

Any assumptions made during this paper were kept to a minimum and were detailed fully here.

(i) As was mentioned above, there was a difference between manipulator navigation and AGV navigation. This paper did not examine manipulator navigation at all. See 1.2.2 for further information on this point.

(ii) There were one or more obstacles in the domain with arbitrary shapes and at arbitrary locations. Some of these obstacles are stationary and others are mobile. The navigation software which controlled the AGV has no prior knowledge of either the number, shape or mobility of obstacles it might encounter while travelling to the goal. Mobile obstacles travelled in straight lines and initially travel in random directions. If a impending collision was detected, the mobile obstacle chose another direction at random. This procedure was repeated by every mobile obstacle in the domain at every collision.

No attempt was made to use existing techniques, such as [KYR92], to predict the intended path of a mobile obstacle. The problem was that these predictions were computationally expensive and, more importantly, they failed in scenarios where random movements by the obstacles were permitted - something which humans walking through an automated warehouse were quite likely to do.

(iii) Mobile obstacles could not travel any faster then the AGV. This meant that there was a known velocity which the obstacle could not exceed. If the AGV could travel at a maximum velocity of one unit per time slice, then all the obstacles in the domain could travel at velocities up to or equal to, but not greater than, one unit per time slice. Some of the algorithms reviewed later in this paper relied on an obstacle to continue moving at the same velocity or in the same direction and these techniques behaved badly if the obstacle changed direction or velocity. These techniques were mentioned briefly for the sake of completeness, but were then excluded from further consideration. All remaining techniques could cope with occasional random changes of direction, as well as of velocity, although this meant that the AGV must never enter within the radius of the safety distance to any object, as the obstacle might randomly change direction at that moment and cause a collision. The algorithms under scrutiny did not attempt to differentiate between permanently stationary obstacles in the domain, and mobile obstacles which had randomly changed their velocity to zero. Because of this all obstacles, both mobile and stationary, were enclosed within a safety margin which the AGV never entered.

(iv) All navigation algorithms were evaluated as to their performance in a two dimensional domain, over a period of time. The time factor is an important one as it was required for representing mobile obstacles. Some techniques were unable to deal with mobile obstacles in the domain at all. They have been briefly mentioned for completeness, but then discounted as being inadequate for our purposes.

(v) The size of the AGV depends on the path planning technique being used. All general purpose AGV navigation techniques generalise the shape of the AGV to one extent or another. This is done so that the same technique can be used with more than one physical model of an AGV. Some techniques represent the AGV by an enclosing circle or rectangle and then find a path for this shape through the domain. Other techniques insist that the AGV must be of point size. Obviously an AGV cannot actually be of point size, but it is possible to represent the AGV as being point sized, and these navigation techniques only look at how the AGV is represented. To do this, place the AGV in an enclosing circle. Then calculate the radius of this enclosing circle and increase the size of every other object in the domain by this amount. Simultaneously reduce the size of the AGV by the same amount. The result is a point sized AGV and objects in the domain which are correspondingly larger.

(vi) All of the navigation techniques which were investigated assumed that the robot had no trailing cables. This meant that the AGV had to be independently powered and that any required computing power was either installed on-board or communicated via any wireless media (i.e. radio, infra-red, television, etc.) Both these conditions are very important. If the AGV had trailing cables connecting to a fixed power/communications port, then the entire nature of the problem changed radically. The problem then become a channel routing problem, rather than a path planning problem. Readers who

are interested in greedy, manhattan, river and other channel routing algorithms are referred to [BAK84], [BRA84], [BUR86], [HUT85], [SHI86], [ULL88], [YOS82]. Some work on parallel algorithms in this area was carried out in [FUN92].

At the risk of stating the obvious, we point out that while some of the researched techniques on path planning were originally intended for use in circuit routing and VLSI design, the subject of circuit layout was totally ignored as it was not relevant to the navigation of untethered AGVs.

(vii) The goal in the domain was of relative point size and was stationary throughout the simulation. Mobile obstacles which encountered the goal by chance regarded it as just another obstacle, stopped before collision actually occurred, and changed direction at random to one which offers movement without collision.

(viii) Only techniques which guaranteed to find a path to the goal if it existed were of interest. Ideally, we would also like this path to be the shortest path to the goal. However, paths which are not *exactly* the shortest, but were still reasonably short were tolerable. However, note that algorithms which may or may not have found a path to the goal even though valid paths did exist were of no interest to us. Also algorithms which may find a very long path to the goal, when there were significantly shorter paths available were of little practical use.

(ix) At the outset of this project, there was a strong requirement that any simulators written be ported to different computer platforms. Target platforms included Dos based 80386 PCs, Unix based Risc workstations and PC-hosted Inmos transputer systems. Therefore all code had to be written in a realistically portable manner - being theoretically portable simply was not adequate. This had two main implications.

Firstly, all programs had to be written in a platform independent high level language which was available on all platforms. The only language available on all intended platforms was 'C', and only some of these were ANSI compatible. To ensure that the same code would compile on all these various compilers, we erred on the side of extreme conservatism when writing any programs. This radically increased the portability of the code, although at the cost of some non-optimised code.

Secondly, no machine specific functionality could be used. Potentially useful, but platform specific instructions for floating point co-processors, virtual memory, graphical user interfaces and the like were not used. Utilising any of this extra functionality would be guaranteed to cause problems and

incompatibilities when porting to a different platform where that functionality was not provided. Therefore, only 'C' routines which worked on all of the intended platforms were acceptable for use.

Original attempts to develop this set of programs in C always worked when run from within the debugger, but crashed intermittently otherwise. After considerable effort spent tracking down this problem, it was discovered that the C compiler had several undocumented bugs in how it coped with far memory pointers. As these far memory pointers were needed to model the domain, this compiler was no longer suitable and had to be replaced. We took this opportunity to tidy up the design of the testbed programs, and rewrote them, this time in C++. This had two benefits. Firstly, the C++ compiler did implement far memory pointers properly, thus solving the original problem. Secondly, the more modular structure of C++ enabled us to more easily isolate the standard shared functionality of the testbed programs and thus considerably enhanced overall code legibility.

Because these coding guide-lines were followed, the testbed programs developed during the course of this paper were easily and successfully ported to a Unix based Risc workstation. However, we could no longer port the testbed programs to the Inmos transputer system, as there was no C++ compiler for that platform.

2. Existing Path Planning Algorithms

2.1. Introduction

Several different existing path planning algorithms were reviewed in this chapter and their relative performances compared. Each algorithm was evaluated on the basis of it's performances in the following areas:

(i) The ability to find a path from the current location of the AGV to the goal, when the dimensions and locations of all obstacles in the domain were known and all obstacles were stationary.

(ii) The computational time required to find this path to the goal, given that all obstacles in the domain were stationary.

(iii) The extent to which the algorithm found the optimal path to the goal, given that all obstacles in the domain were stationary. An optimal path may be the one where the AGV was required to travel the shortest distance, or it may be the one where the AGV reached the goal in the shortest period of time. It may even be the path that stayed the furthest from the other obstacles in the domain.

(iv) The ability to find a path from the current location of the AGV to the goal, when the dimensions and locations of all obstacles in the domain were known and some or all obstacles were mobile.

(v) The computational time required to find this path to the goal, given that some or all obstacles in the domain were mobile.

(vi) The extent to which the algorithm found the optimal path to the goal, given that all obstacles in the domain were mobile. As in (iii) above, an optimal path may be the one where the AGV was required to travel the shortest distance, or it may be the one where the AGV reached the goal in the shortest period of time. It may even be the path that stayed the furthest from the other obstacles in the domain.

Routing is the general term for deciding how to place wires from one block (i.e. chip, transistor, etc.) to another block on the same Printed Circuit Board (PCB), or in the same VLSI chip. This is a well defined area of research and it is broken into two main categories: Channel Routing and Circuit Routing.

All works in both categories assumed that the object being routed (a wire or an AGV) was of point size. Channel Routing algorithms, which assumed no obstacles in the domain, were not suitable and were omitted from further discussions. Circuit Routing algorithms, however, assumed that the size and location of any obstacles in the domain were known and that all obstacles which were present in the domain were stationary. They would then deal with finding paths from a point on the side of one chip to

a point on the side of another non-adjacent chip. As the target point was not on an adjacent chip, there was usually one or more obstacles which needed to be circumnavigated. However, several different Circuit Routing algorithms existed, with different priorities and different degrees of success for different scenarios. Some of the more common approaches are outlined below.

2.2. Map Sectors

This algorithm required that the entire domain to be traversed be marked off into sectors, whose size was such that the AGV is of relative point size. Each sector was identified by it's co-ordinate relative to some absolute location and marked as either traversable (free space) or non-traversable (obstacle). Because of this, the starting and ending locations were simply two sets of co-ordinates in the domain.

Once the entire domain was divided into sectors and categorised, the idea was to travel from the sector containing the start position to the sector containing the goal position via a series of traversable (free) sectors. Implementation was relatively straightforward, but there were two main limitations to this approach.

(i) If any part of a sector was blocked by an obstruction, then the entire sector was marked as non-traversable. While this simplification may not seem like a serious one, it meant that in environments where two obstacles were close together, but with a potentially valid path between them, the two obstacles could be marked as one large obstacle, and the path between them would be obscured. In environments which had large areas of open space in them, this was not a serious problem, although it may cause the AGV to choose a path which was longer than the absolute shortest path. In more cluttered situations, however, this may have prevented the AGV from being able to find any path to it's destination.

    This problem can be solved by increasing the resolution of the domain and therefore reducing the size of the sectors used. However, this also dramatically increases the number of sectors in the domain and therefore the time required to find a path to the goal. Reducing the size of the sectors also means that the AGV is no longer of relative point-size. As all these map sectoring algorithms depend on the AGV being point-sized, the AGV must somehow be shrunk so that it is still of relative point size. This is done by expanding every one of the obstacles in the domain by the radius of the AGV and simultaneously reducing the size of the AGV to a point.

(ii) This algorithm was not designed to deal with mobile obstacles. Because this algorithm was originally designed for PCB work, all obstacles were assumed to be stationary. Thus, when an

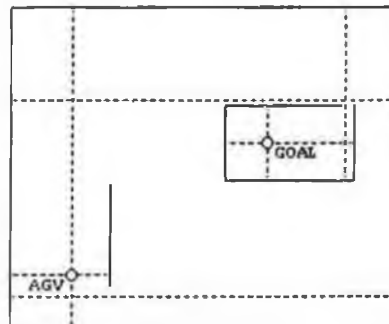
obstacle was first detected, the sector it was in was marked as permanently non-traversable. This means that although an obstacle may move to another sector, the now free sector was still considered non-traversable, and a sector which was considered to be traversable was actually occupied.

The only way in which mobile obstacles could be dealt with was to totally re-categorise all of the sectors in every time slice, and so maintain a true up-to-date representation of the domain in the mapped sectors. However, this was a very time consuming solution.

Storing time as a third dimension for each sector would cope with mobile obstacles by changing the problem into one where there was a series of static 2 dimensional problems to be solved. Attempts to implement a three dimensional map of sectors, with time as the third dimension, worked to limited success. This "Hierarchical Strategy" is outlined later in this chapter.

Once the domain was marked out into traversable and non-traversable sectors, there were several different methods available for calculating the route from the starting point to the goal. The most relevant of these are outlined below.

2.2.1. Hightower



This algorithm ran vertical and horizontal lines of point width from both the start and end locations in the domain until they reached an obstacle in each direction, or the end of the domain. These lines were stored in two sets - a set of lines which originated at the goal and a set of lines which originated at the starting location. Every point on each of the lines in the two sets was then checked to see how long a perpendicular line could travel before reaching an obstacle. The perpendicular line which would be the longest was then inserted into the relevant set and the two sets of lines re-examined to find the new longest perpendicular line. This procedure was repeated until a line from one of the sets intersected with a line from the other set. When this happened, the algorithm had found a path to the goal with the minimum number of corners (or "vias" in VLSI terminology).

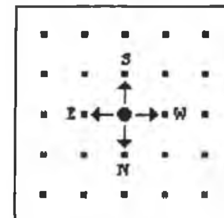
While this algorithm was quite fast for very simple sparse scenarios, performance quickly deteriorated in more cluttered scenarios. If the scene was a complicated one, this algorithm did not guarantee to find a

path, even if one existed [HIG69], [SOU81]. Modifications to this algorithm [HEY80] improved it's performance somewhat, but it still did not guarantee to find a path to the goal, even if one existed.

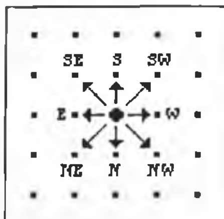
2.2.2. Lee's Algorithm

Lee's algorithm is a flexible and robust algorithm which was originally intended to deal with the routing of a wire from a start location to an end location on a PCB, navigating around chips and any other obstacles which may block the direct route.

Each point adjacent to the start location is examined to see if it is the goal, or if it is a barrier (i.e. any part of an obstacle). If neither, then the square is assumed to be traversable and is marked as having been examined. The direction the square was examined *from* is also stored. The process is then repeated for each of the adjacent squares, causing a ripple effect to spread out across the domain. This ripple effect will eventually search every square in the domain, even around obstacles and reach the goal. Because every square is marked with the direction it was examined from (i.e. it's predecessor), it is possible and quite easy to backtrack from the goal to the starting point, thus generating the path to be traversed by the AGV to reach the goal. Not only does this algorithm guarantee to find a path if one exists, but it also guarantees to find the shortest possible path to the goal. [BEL87]



One flexibility of Lee's algorithm is that the definition of what exactly is an adjacent square is flexible. In most circuit routing applications of this algorithm, every point has only four adjacent squares to each square. However, the algorithm works equally well with eight adjacent squares to every square and this is more suitable for our purposes of navigating an AGV.



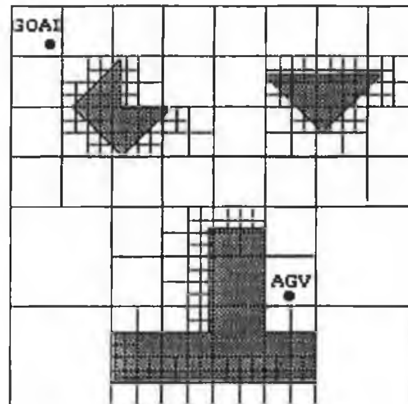
Unfortunately, if the size of the domain is large, or the resolution high (or both) then the memory requirements grow considerably while the process itself becomes quite cumbersome and slow. Theoretically, this algorithm takes time in proportion to the square of the distance between start and goal sectors. [SOU81]

Several variations of Lee's algorithm have been developed to try and reduce the time required for path finding. Rubin's [RUB74] and Soukup's Arrow [SOU78] variations both make use of the fact that the algorithm knows where the goal is, and give preference to points which are deemed to be closer to the goal. These are 20 to 50 times faster than the original unmodified Lee's algorithm and go some way towards solving the performance issue.

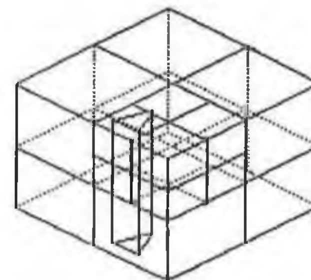
2.2.3. Straight Line

The simplest way to plan a path to the goal is to draw a straight line from the current location of the AGV to the location of the goal. If the straight line travels through sectors which has been deemed non-traversable, then adjacent sectors are recursively checked for traversability, using Lee's algorithm, until a route has been found back onto the original straight line path. Travel continues along the straight line until the goal is reached. or another obstacle is encountered. While this approach does use Lee's algorithm for part of the search, it is quicker than using Lee's algorithm overall. This is because the distance from one side of an obstacle to another is typically significantly shorter than the distance from the start to the goal and thus the time taken by Lee's Algorithm to find the other side of the obstacle is significantly less than the time taken to find the goal. This approach has been used by the Jet Propulsion Laboratories [THO77] as a basis of a navigation system for their AGV project, which is intended for planetary exploration work.

2.2.4. Hierarchical / Octree



The Hierarchical approach can be used as an alternative to the Map Sectoring approach for two dimensional domains with only static obstacles in the domain. In an attempt to cope with mobile obstacles, this Hierarchical approach uses time as a third dimension when categorising every sector in the domain. Thus, every node is referred to by it's 2D co-ordinates and the time t. This is referred to as it's location in "space-time". The idea is that all obstacles, whether static or mobile in normal space, will appear static in space-time. Each sector is marked as traversable or non-traversable for a a particular time period. This is based on the concept that the AGV needs to know the sector is free only when the AGV tries to traverse it and constantly rechecks the traversability of all sectors near the AGV. Basically, the idea is to find a path from the source to the goal through cubes which are empty *at the time they are being traversed.* [FUJ89]



A hierarchical structure was used, where space-time for the entire domain was repeatedly divided into 8 cubes of equal size. Each cube in turn was sub-divided into 8 smaller cubes. This process was repeated until a cube fitted into one and only one of these categories:

(i) cube was totally empty
(ii) cube was totally within the projected path of the obstacle.
(iii) cube contained part of the projected path of a vertex of an obstacle.
(iv) cube contained part of the projected path of an edge of an obstacle.

It was a reasonably fast technique, and certainly faster than rechecking the entire domain in every time slice. Due to it's ability to sub-divide cubes into one of four categories, resolution increased where it was needed most - at the boundaries between an obstacle and free space. This meant that an obstacle could be of any shape and the exact shape was stored, not an enclosing rectangle. Safety margins could also be easily implemented, as all obstacles were simply expanded by that distance.

However, there are three serious weaknesses with this approach:

(i) There was no guarantee that a path to the goal would be found, even though one may exist.
(ii) Movement by the obstacles were limited. Translation movement were allowed, but any rotation movement by any of the obstacles in the domain required the entire hierarchical tree of cubes to be totally re-built.
(iii) Calculations required to find a path to the goal could easily reach combinatorial explosion levels and therefore increased the time required to intolerable levels.

2.3. Graph Theory

The basic idea behind this approach was to represent the entire domain in a mathematical matrix, and to then use mathematical techniques to find the least expensive route through the matrix. This is done by marking all the important locations in the domain (including the starting and goal locations) as elements in a two dimensional matrix. These locations are called "nodes". All nodes which can directly connect to each other are called "Adjacent Nodes". The cost of travelling from one node to another is stored in an "Adjacency Matrix" and is called an "arc" or "edge". There are several different approaches available for deciding which are the important locations in the domain and two of the more important approaches are discussed in more detail below.

The problem of finding the shortest path from starting location to the goal was now simply a problem of finding the cheapest path through a given network. Several different techniques which can be used to find the cheapest path through this network to the goal are discussed below. These techniques are all established network searching algorithms and are entire areas of research in their own right. In a later chapter, the different algorithms have been implemented and are compared against each other for

effectiveness in the sample domains. It should be noted that all of the following graph theory techniques outlined below are guaranteed to find a path to the goal if one exists.

It is important to explain the difference between graph trees and graph networks before we proceed any further. In a graph network it is possible to travel in a cyclical manner - to travel down through the network and eventually loop back to a node previously visited. This is simply not possible in graph trees, due to the underlying design structure of trees. Unless this looping is being checked for, any graph tree algorithms which are unwittingly used on a graph network may loop forever. A simple amendment to existing tree searching algorithms prevented this looping from occurring. The solution was to mark each node as visited and to store the cost of reaching that node with the other node information. When the traversal algorithm visits a node, it first checks if the node has already been visited. If it has already been visited, the current cost of reaching the node is compared against the previous cost of reaching the same node. If the current visit has less of a cost then the previous visit to this node, the algorithm would overwrite the previous visit with the current visit and continue as per normal. However, if the current cost was greater than the previous cost, the current visit was abandoned and the path travelled to that node trimmed out of the list of possible paths which might have reached the goal. This trimming of the search space prevents graph tree traversal algorithms from looping when used in graph networks. Incidentally, by only using the shortest path to a given node, it also makes the algorithm more efficient. [WIN84]

An advantage of graph theory was that obstacles could be of any shape and their exact shape was stored, not an enclosing rectangle. Some techniques [KHE88] require that the obstacles be enclosed in rectangles or circles to simplify processing. Doing this also eliminated some valid paths and was therefore undesirable. This was not required for either of the graph theory approaches.

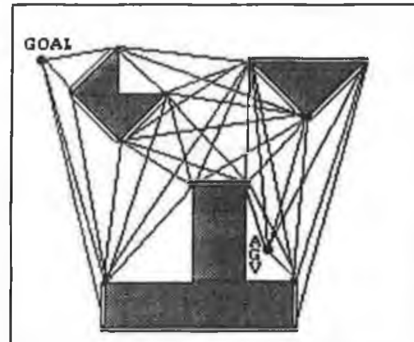
While conventional graph theory techniques assume that all obstacles present in the domain are stationary, some extensions to these techniques have been reported which allow these techniques to cope with mobile obstacles. These existing extensions are discussed below. In Chapter 4, we detail another extension which we developed to allow mobile obstacles in the domain.

2.3.1. Visibility Graph

This is sometimes also called the "Delaunay Triangulation", although this term is usually used in conjunction with Voronai Diagrams.

For every obstacle, the extremities are found, and the free locations immediately adjacent to these extremities are considered to be the important locations in the domain and are stored as nodes. The goal

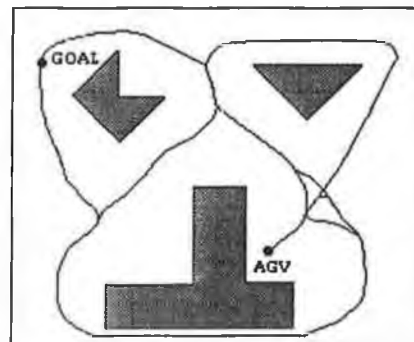
and starting locations are also considered to be important locations and are also stored as nodes. All nodes which can be connected by a straight line are deemed to be "line of sight" visible to each other and are connected by a common edge.



The rationale behind this idea is that the shortest distance between two points is a straight line. If the goal can be reached in a straight line from the current position (i.e. the goal is "visible") then the AGV should travel that path directly to the goal. If there is an obstacle blocking the path to the goal, the shortest route around the obstacle is for the AGV to travel in a straight line to one of the obstacles "visible" extremities. Once there, the AGV can try again to travel in a straight line to the goal, or in a straight line to the visible extremities of whatever is in the way.

2.3.2. Generalised Voronai Diagram

A Generalised Voronai Diagram (GVD) is the locus of points which are equidistant from any adjacent obstacle boundaries, including the domain boundaries. [SED90] In other words, it is the collection of points which are in the middle of every free space in the domain.



There were two steps involved in building a GVD for a domain. Firstly, a complete Visibility Graph was built for the domain in question. The mid-point for every edge within that graph was then found. Collectively, these mid-points are called the GVD for the domain.

There are two main disadvantages to representing the domain using a GVD. Firstly, it is much more complicated to build than a visibility graph. As was mentioned above, the initial requirement for generating a GVD is a completed Visibility Graph. Secondly, the path generated when using a GVD is almost always longer then one generated when using a Visibility Graph. This is because the points along which the path is plotted are always in the middle of any free space. This difference becomes significant when the AGV has to turn a corner. The Visibility Graph approach would let the robot travel as close as possible to the corner vertex without actually colliding. The GVD approach would force the robot to turn the corner in the middle of the available free space - usually a good distance away from the actual corner.

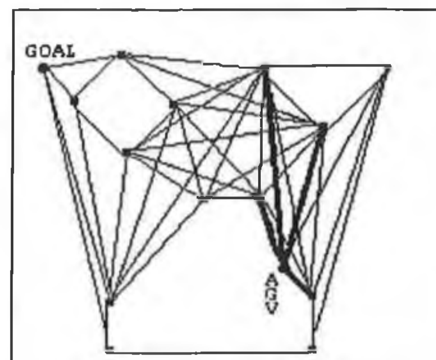
The path found using the GVD was always the "safest" path for the robot to travel to the goal, as all the paths always stayed as far as possible away from all of the obstacles in the domain. Furthermore, in situations where a minimum safety margin was required, all obstacles were expanded by that amount before the GVD was calculated. This was important for scenarios where the consequences of a collision would be severe and far outweigh the cost of the additional travelling time required by following a slightly longer path. (i.e. automated nuclear fuel carriers, automated munitions vehicles, etc.).

Another advantage of the GVD algorithm is that it has been successfully amended to allow the navigation of a non-point sized AGV. In one report, [TAK89] the authors used a rectangular AGV with a pair of steering wheels at the front, and a pair of driving wheels at the rear. The initial path planning was accomplished using GVDs and standard graph theory. However, deciding to make a rectangular AGV follow this intended path required some extra functionality. Fuzzy logic was used to decide at what point the front wheels should turn and to what degree, so that the rear wheels would trail safely past the obstacle (similar to how a bus driver knows how to turn a corner in a bus). Their report also discusses how they modified the corners in the path, so that the corners were gentle curves rather than sharp and impractical.

In another report [SUH88] the authors designed another variation on GVDs which allows the user to specify a minimum safety distance between a non-point-size AGV and any obstacles. It also for allows some optimising of the arcs which the AGV would have to follow if it followed the GVD exactly. The routes that these techniques produce, while still not being the absolute minimum distance to the destination, are shorter then the original GVD specified route. Also, the route now being travelled is more realistic, as the AGV now gently curves around corners, rather then taking sharp turns just because the obstacles do.

2.3.3. Breadth First Search Algorithm

The idea behind the Breadth First search algorithm was similar in a way to Lee's algorithm. From a given starting node, such as the AGV's current location, all adjacent nodes were examined to see if any of them was the goal. If not, all the nodes adjacent to all these nodes were then examined. This ripple spread out through the entire network of nodes until the goal was reached.



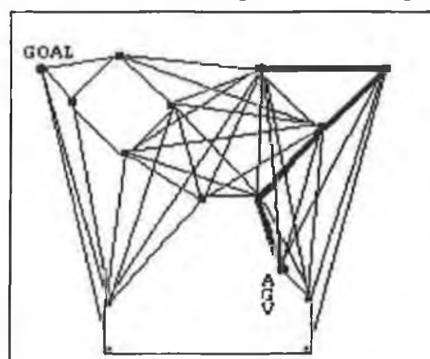
If a valid path to the goal existed, this search algorithm was guaranteed to find it and to always find the path with the lowest cost first. If the cost of travelling between a given pair of nodes was proportional to the distance between those nodes, the path to the goal with the lowest cost was also the shortest path to the goal. Note that while this algorithm was originally developed for use on graph trees, it could also be used in graph networks once the modifications outlined in 2.3 above were implemented. In pseudo code, the modified algorithm was as follows:

(i) Insert the start node (i.e. the starting location of the AGV) into an otherwise empty queue.
(ii) If the queue was empty and therefore no node could be made the current node, go to (vii).
(iii) Take the first element from the queue and make it the current node.
(iv) If the current node had been visited before, and the previous cost of the visit was less than the current cost of the visit, dispose of the current node and go to (ii).
(v) If the current node was the goal, mark the node as visited, dispose of the current node and go to (ii).
(vi) If the current node was not the goal, mark the node as visited. Then append all nodes adjacent to the current node to the end of the queue, dispose of the current node and go to (ii).
(vii) If the goal node has been visited, display that the goal has been found. Else display that the goal has not been found.
(viii) End program.

Note that when a vertex node was inserted into the fringe list, it was always placed at the end of the fringe list. This meant that all other nodes already in the fringe list would be visited before this new node was visited, thus forcing a Breadth First search of the graph network.

The running time required by the Breadth First algorithm to find the goal depended on the density of the graph. The number of steps required for sparse graphs was in proportion to the number of nodes in the network summed with the number of edges in the network. By contrast, the number of steps required for dense, or complete, graphs was in proportion to the square of the number of nodes present in the graph [SED90].

2.3.4. Depth First Search Algorithm



The idea behind the Depth First search algorithm was to search the network on a recursive basis, optimistically explored a particular direction to exhaustion, before it would search in another direction for the goal. A node adjacent to

the starting node was checked, and all of it's adjacent nodes checked to exhaustion, before the next node adjacent to the starting node would be checked.

If a valid path to the goal existed, this search algorithm was guaranteed to find it. However, in it's unmodified form, this algorithm returned the first path found to the goal, not necessarily the shortest path available. To ensure that this algorithm did return the shortest available path to the goal, two modifications were made. Firstly this algorithm had to be made suitable for graph networks, rather than graph trees, as outlined in 2.3 above. Secondly, the search loop should only terminate when the queue was totally empty, not just because the goal had been found for the first time. In pseudo code the modified algorithm was expressed as follows:

(i) Insert the start node (i.e. the starting location of the AGV) into an otherwise empty queue.

(ii) If the queue was empty and therefore no node could be made the current node, go to (vii).

(iii) Take the first element from the queue and make it the current node.

(iv) If the current node has been visited before, and the previous cost of visiting was less than the current cost of visiting, dispose of the current node and go to (ii).

(v) If the current node was the goal, mark the node as visited, dispose of the current node and go to (ii).

(vi) If the current node was not the goal, mark the node as visited. Then insert all nodes adjacent to the current node to the beginning of the queue, dispose of the current node and go to (ii).

(vii) If the goal node has been visited, display that the goal has been found. Else display that the goal has not been found.

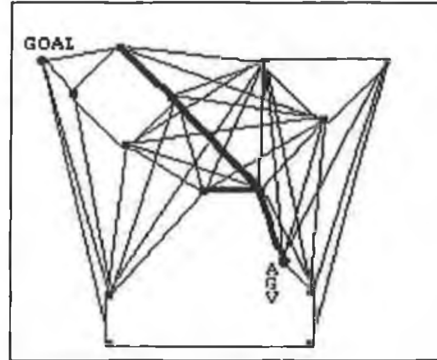
(viii) End program.

Note that when a vertex node was inserted into the fringe list, it was always placed at the beginning of the fringe list. This meant that all other nodes already in the fringe list had to wait until after the new node was visited before they were visited. The effect of this insertion technique was to force a Depth First search of the graph network.

Like the Breadth First search algorithm discussed previously, the number of steps required by the depth-first algorithm to find the goal depended on the density of the graph. The number of steps required for sparse graphs was in proportion to the number of nodes in the network summed with the number of edges in the network. By contrast, the number of steps required for dense, or complete, graphs was in proportion to the square of the number of nodes present in the graph [SED90].

2.3.5. Priority First Search Algorithm

The idea behind the Priority First search algorithm was to search the network, always checking the nearest node first. This was basically a variation of the Breadth First search algorithm. The cost of reaching each of the unvisited nodes adjacent to any of the visited nodes (initially just the starting node) was calculated, and the unvisited node with the cheapest cost was visited. This was repeated until the goal node was visited.



If a valid path to the goal exists, this search algorithm guaranteed to find it and guaranteed that the path found was the shortest. As with the previous two algorithms, this algorithm must be made suitable for use with graph networks, rather than graph trees. This requires the modifications outlined in 2.3 above. In pseudo code the modified algorithm would be expressed as follows:

(i) Insert the start node (i.e. the starting location of the AGV) into an otherwise empty queue.

(ii) If the queue was empty and therefore no node could be made the current node, go to (vii).

(iii) Take the first element from the queue and make it the current node.

(iv) If the current node has been visited before, and the previous cost of visiting was less than the current cost of visiting, dispose of the current node and go to (ii).

(v) If the current node was the goal, mark the node as visited and go to (ii).

(vi) If the current node was not the goal, mark the node as visited. Then insert all nodes adjacent to the current node into the queue. The nodes inserted into the queue are inserted in ascending sequence, using the total cost required to reach them as the key field. Finally, dispose of the current node and go to (ii).

(vii) If the goal node has been visited, display that the goal has been found. Else display that the goal has not been found.

(viii) End program.

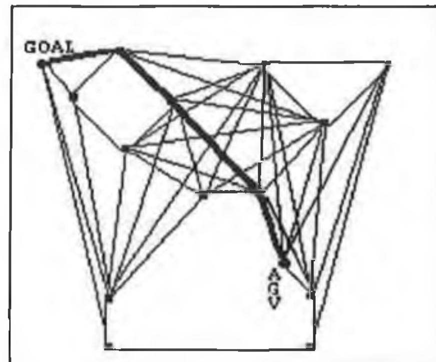
When a vertex node was inserted into the fringe list, a rough approximation was made of the remaining distance between the vertex node and the goal node. This approximate distance was used to decide the insertion point of the node into the fringe list. Those vertex nodes which were nearer the goal were nearer the front of the list, while those further away from the goal were towards the end of the list. The effect of this insertion technique was to force a modified depth-first search of the graph network. The reasoning

behind this modification was to prevent the search from going past the goal and wasting time searching some remote corner of the network - a weakness of the Depth First search.

This algorithm will find the goal in less than proportion ((E+V) log V) steps (where E was the number of edges and V was the number of vertices, or nodes, in the domain) if the graph was a sparse one, and V^2 steps if the graph was a dense one [SED90].

2.3.6. A* Search Algorithm

The A* search algorithm also sorted the nodes which were eligible for checking and checked the node with the lowest cost. However, the cost was calculated in a very different manner to the Priority First search. When a vertex node was being inserted into the fringe list, a rough approximation was made of the remaining distance between the vertex node and the goal node - but it had to be an underestimate. This approximate distance was summed with the total distance travelled so far to reach the current node. This summation



node. This summation gives a relative index. Vertex nodes were then inserted into the fringe list in ascending index order. Those vertex nodes which are closer to the idealistic straight line to the goal would be nearer the start of the list, while those further away from the straight line path to the goal will be towards the end of the list. The effect of this insertion technique was to force a modified depth-first search of the graph network. The cost required to reach a node was added to the estimated cost of reaching the goal from this node to give the actual cost for the node. The node with the cheapest cost was then visited, and if it was not the goal, all adjacent nodes were costed, inserted into the queue and the process repeated.

If a valid path to the goal existed, this search algorithm was guaranteed to find it and guaranteed that the path found was the shortest. However, note that for this to be true, the estimated cost of reaching the goal from a node must be an underestimate. Overestimating will still allow the path to be found, but the path may not be the shortest. As with the previous search algorithms, this algorithm had to be made suitable for use with graph networks, rather than graph trees. This requires the modifications outlined in 2.3 above. In pseudo code the modified algorithm would be expressed as follows:

(i) Insert the start node (i.e. the starting location of the AGV) into an otherwise empty queue.

(ii) If the queue was empty and therefore no node could be made the current node, go to (vi).

(iii) Take the first element from the queue and make it the current node.

(iv) If the current node has been visited before, and the previous cost of visiting was less than the current cost of visiting, dispose of the current node and go to (ii).

(v) If the current node was the goal, mark the node as visited and go to (ii).

(vi) If the current node was not the goal, mark the node as visited. Then insert all nodes adjacent to the current node into the queue. The nodes inserted into the queue are inserted in ascending sequence, using the total cost required to reach them summed with the lower bound estimated cost of reaching the goal as the key field. Finally, dispose of the current node and go to (ii).

(vii) If the goal node has been visited, display that the goal has been found. Else display that the goal has not been found.

(viii) End program.

The reasoning behind this modification was to try all paths which are as close as possible to a straight line to the goal. Only when all those paths have been exhausted, would this approach try searching further afield. This prevented the search from going past the goal and wasting time searching some remote corner of the network - as was possible with the Depth First search. It also trimmed down on the number of steps required to find the goal node.

2.3.7. Collision Avoidance in Graph Theory

While GVDs were initially intended for work with stationary obstacles, there were two approaches found for using GVDs in conjunction with mobile obstacles.

(i) The first approach had additional software functionality available which could take over in the case of an impending collision. To date, I found two reports worth mentioning where collision avoidance functionality was added which would overrule the GVD navigation software when an imminent collision was detected.

One report [GRI90] avoids mobile obstacles simply by changing the velocity of the AGV but while still keeping to the original path specified in the GVD. Initially, an unmodified network traversal algorithm planned a route from the AGV's current position to the goal. This path would avoid collisions with all known (and assumed stationary) obstacles. If, while travelling on this path, a mobile obstacle was detected within a certain safety distance of the AGV, then a impending collision was declared. Collision avoidance software would then suspend the path following software and decide by how much the AGV's velocity should be increased or decreased in order to avoid collision. When the AGV was deemed to have avoided the obstacle, the collision avoidance

software hands back control to the path planning and following software and the AGV continues on towards the goal. Note, however, that at no stage in this process did the AGV deviate from it's originally planned path.

This non-deviation from the original path was quite important. While non-deviation from the intended path simplified some matters (i.e. at what point should the AGV get back onto it's original path or should a completely new path to the goal be found), it did mean that the AGV could not easily handle the situation where an obstacle approached it head-on. If more then one AGV was independently working in the same domain, each AGV would view other AGV's as mobile obstacles. In this scenario, two identical AGVs which approached each other head-on would both stop on their current paths and wait, in deadlock, for the other to move out of the way first. To us, this was a serious flaw in the technique.

The other reviewed report on collision avoidance software [WAN91] changed both the velocity and direction of the AGV in order to avoid pending collisions. The Least Mean Square Error estimation technique was used to estimate a likely position for a mobile obstacle at time t+1, given it's positions for times t, t-1, t-2,... This technique assumed that previous movements by an obstacle could be used to give an indication as to the next intended move of the obstacle. More recent movements were weighted as more important than older movements. Given this estimated next location for the obstacle, it was possible to decide on a route for the AGV which would best avoid it. In the examples given by the authors, this estimation technique for obstacle movement worked quite quickly. However, this technique fails in scenarios where the mobile obstacle changes direction quickly and without warning, as humans wandering through the domain would tend to do. It is also very computationally expensive.

(ii) The other approach was to constantly rebuild the GVD for every time-slice. This meant that rather than using the one graph to represent both static and mobile obstacles, a new graph is used in every time slice which only represented static obstacles. This meant that all mobile obstacles appeared to be static in a particular time slice. It also meant that normal graph traversal techniques could then be used to find a path to the goal for that time slice. As the location of a "static" obstacle may have changed from one time slice to another, the path from the AGV to the goal had to be recalculated for every time slice.

As the time required to build the entire graph was generally quite significant, this approach did not look promising. However, we investigated the feasibility of updating only the relevant parts of the

graph, in the hope that it would make this approach practical. Our work in this area is covered in detail in Chapter 4 below.

2.4. Landmark Navigation

In the real world, humans navigate with very imprecise knowledge of the domain. Current theories in cognitive psychology indicate that humans tend to navigate using landmarks. It seems intuitive that successful AGV navigation in the real world should also use the same imprecise knowledge of the domain - just like humans do and therefore an AGV which imitated this landmark navigation strategy should be able to navigate in the real world as successfully as humans do.

Before looking in more detail at some work in this area, the term "landmark" should be explained. A landmark is defined as a recognisable feature in the domain. Apart from storing information about the size and location of each landmark, a list of all other adjacent landmarks is also stored. Note that the types of landmarks used are not exhaustive. Only "point-sized" landmarks (i.e. corners, hilltops, etc.) are used in the techniques outlined below. No attempt is made to deal with larger "extended" landmarks (i.e. hedges, roads, ridges, etc.)

The AGV's current location, as well as the destination location are specified relative to the nearest landmark. As all navigation instructions were given relative to the nearest landmark, the idea behind the path planning algorithm used was to build up a list of landmarks which the AGV should pass, from one to another and eventually to a landmark adjacent to the goal. Also, as the AGV travels from the current landmark to the next landmark on it's intended path, it always uses the local co-ordinate system of the current landmark and thus keeps the cumulative error within tolerance levels. Most other navigation techniques use some form of absolute measurement, in which significant cumulative errors can build up over large distances, with potentially serious consequences [THO77]. There were two reports of interest reviewed.

The first report [LEV90] suggested that once a landmark in the domain was acquired, the exact relative range and angle of all adjacent landmarks to the current landmark were estimated *imprecisely* and combined to form the "view frame" for the current landmark. This procedure was then repeated for every other landmark in the domain. After the path was calculated and as the AGV travelled to the goal, corrections to the imprecise measurements for a landmark would be made if the AGV travelled close to that landmark.

The second report [MCD84] also stored landmarks and their location relative to other adjacent landmarks in it's internal representation of the domain. However, this technique handled imprecise information about landmarks in a slightly different manner. All objects in the domain (both obstacles and landmarks) were enclosed in a probability area[1]. This probability area denoted where there was a significant probability that an object was actually located, using all the information available at the time. Rather than store the size and location of an object imprecisely, the precise area enclosing the probable location of the object was stored. As the AGV approached the object, more precise information became available, and the probability area enclosing the object reduced in size, until eventually it was the exact size of the actual object.

Once all the landmarks in the domain were mapped out, both reports navigated through the domain by building a list of landmarks which must be passed when travelling from the current location of the AGV to the goal. When the AGV travelled towards the goal and, en route, reached a landmark, it used that landmark's local view frame to decide the heading needed to reach the next landmark on the path. If however, on reaching a landmark, the AGV happened to find a landmark in the local view frame which was 2 or 3 steps further down the intended path, then it went directly for that landmark, and thus took a shortcut past the intermediate landmarks. This would happen if the extent of the initial measurements were inaccurate enough to prevent the path planning software from finding the optimal path to the goal.

However, there was a problem with these reports - all obstacles must be stationary. If mobile obstacles were allowed in the domain, then the possibility exists that one of the landmarks which the AGV used to navigate by was actually moving at the same time as the AGV! This would have potentially disastrous consequences and must be prevented. Pre-knowledge as to the mobility of the obstacles was not acceptable, as this assumed that all mobile obstacles in the domain were identified to the AGV from the outset, which was not unrealistic. Updating all of the landmarks in every time-slice was a work around to the problem, although time consuming. To keep the landmarks and their view frames up to date required that each of the existing landmarks had to be rechecked to see if it should continue being a landmark, possibly removing some existing "landmarks" and possibly inserting some new "landmarks". After the landmarks were checked for a given time-slice, the view frames for each of these landmarks needed to be rebuilt. The time required to rebuild the domain map for every time-slice did seriously slow down the overall execution time of the path planner.

---

[1] While the authors refer to this probable area enclosing an object as it's "fuzzy coordinates", and the domain containing these "fuzzy coordinates" as a "fuzzy map", there is no relationship between the use of these terms and the term "fuzzy logic". We have used the terms "probability area" and "domain", as they are less ambiguous.

2.5. Configuration Space

This algorithm was intended to move a non-point robot through a 3 dimensional domain containing obstacles [BRO82], [BRO83], [BRO87], [DON87]. There were three distinct parts to this algorithm.

Firstly, the size of all obstacles in the domain were increased to correspond to the decrease in size of the robot to point size. This new representation of the domain was called the "Configuration Space", or "C-Space" and contained obstacles, free space (called free cones and freeways) as well as the point sized robot. Mobile obstacles were represented in this domain by updating the free cones and freeways after every time-slice.

Secondly, all plausible movements for the robot at it's current location within the C-Space were found. This was done by having several different processes (called "Local Experts") which, given the current position of the robot within the domain, would independently try to generate what they thought the next position for the robot should be. As each Local Expert employed a different strategy to navigate the robot around the obstacle, each Local Expert returned a different result to the higher level navigation program.

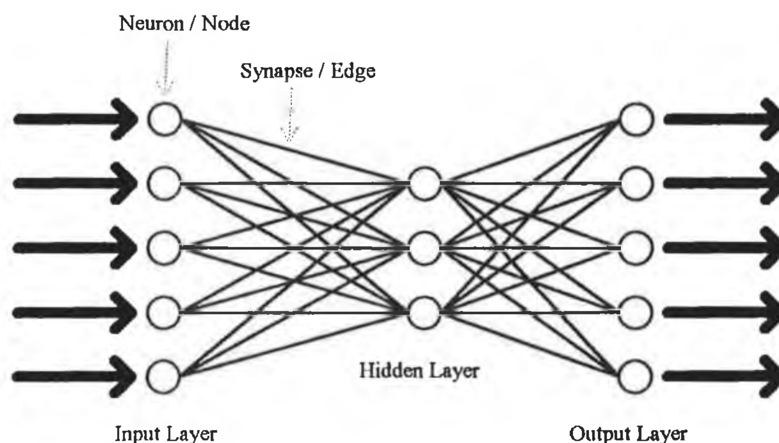
Thirdly, a higher level navigation algorithm accepted all the proposed movements from the Local Experts, and decided which of the suggested moves was the best one for the robot to take to reach the goal. While several different algorithms could be used for this, the simplest one was the "Bumble Strategy" - where by exhaustive, brute force searching, the goal was eventually found.

A sophisticated enhancement was developed by Brooks and Lozano-Peréz [BRO87] to improve the robot's navigation between obstacles. The C-Space technique would usually expand the size of all the obstacles in the domain by the distance from the centre of the robot to it's furthest extremity and at the same time reducing the size of the robot to a point. The algorithms would then navigate a point (the robot) around these artificially inflated obstacles. While this approach works most of the time, it has weaknesses. For example, it did not work in situations where a *non-symmetrical* robot, by turning sideways, could fit through a gap that was otherwise impassable. The enhancement was to increase the size of the obstacles in the domain by the size of the robot *in that dimension* while reducing the robot to point size. This enhancement allowed the robot to rotate and fit down some narrow gaps which would otherwise have been considered blocked.

2.6. Neural Networks

While simple neural networks have been around for some time (they were called "Perceptrons" in the 1960s), all work with them was dropped after a critical report [MIN69] undermined confidence in the technology. In more recent years, neural networks re-established themselves as very useful, quick and robust techniques to use for pattern recognition. There was a considerable volume of mathematical research on the different types of neural networks available, explaining how they work, and the benefits of using the different learning algorithms. For a detailed explanation of this complex area, interested readers are referred to [KOS92], [LAW90] and the series [MCC88A], [MCC88B] and [MCC88C], which provided both detailed theory as well as sample programs. A good introduction to this area was provided by [ALE83] and [JOH88]. Some basic concepts and terminology should be explained below before proceeding any further.

The design of a neural network was supposed to resemble the structure of the human brain. The human brain was made up of neurons interconnected by synapses, which a neural network emulated using nodes and edges.



However, a neural network was only usable after it has been programmed (or "trained") for the intended application domain. This was done by building the neural network structure, such as the model shown, and initialising the weights (or strengths) for each of the edges. Most neural networks were built of an input layer, (which received the inputs to the network from the outside world), and an output layer, (which output the conclusions of the neural network to the outside world). Between these two layers can be none, one or more layers of so-called hidden nodes. The network was then shown an input to be recognised, which was placed on the nodes in the input layer. All other nodes in the network were initialised. The value of each node on the input layer was combined with the weight on one of it's edges, and the result added to the value for the node at the other end of that edge. This was repeated for every edge on every node on the input layer. At this point, each node on the next layer (usually the hidden layer) had accumulated some value. This procedure was repeated for every node on that layer, and repeated in this fashion for every layer in the network, until the values for the nodes in the output layer were calculated.

Obviously, the weights on the various edges were important to the accuracy of the output. The most popular method used to determine the weights needed on the edges was "Back Propagation". This technique initialised all the weights to a random value and then showed an input value to the neural network, and compared the generated output with the desired output. The differences were analysed and the weights of the edges which produced this incorrect output modified slightly so as to reduce the inaccuracy by a small amount. This error reduction was repeated (or propagated) backwards for every layer in the network, until the input layer was reached.

The entire procedure of presenting inputs, generating and comparing outputs and modifying the relevant weights as necessary, was repeated for several iterations, until the generated output was within tolerance levels of the desired output. The neural network was then considered trained, and the "Training Phase" ended. Note that a network trained with just one pair of input-output results would result in a network which could recognise just that one input accurately and generate the corresponding output accurately (and with relatively little training). To have a network which can correctly deal with many different input patterns required that the network be initially trained on a set of many different input patterns known as the "Training Set".

There were several problematic aspects to implementing neural networks. The size of the "Training Set", ensuring that the data in the Training Set was representative of the domain in general and the order in which these different training patterns were presented to the network during training were still done on an trial-and-error basis. Other weaknesses in current neural network technology lie in deciding how many hidden layers were required, how many nodes should be in each of the layers and how these layers should be interconnected. As there were no rules or guidelines established yet on designing neural networks, these questions could only be answered by trial-and-error, combined with a knowledge of what worked before in other scenarios.

The neural network would be built and then trained for a few thousand training cycles using the Training Set. At the end of this training session, if the network had not converged (i.e. if the difference between desired and actual outputs showed no sign of reducing) then that particular combination of neural network, with it's nodes in each layer and their interconnections were abandoned. Another neural network would then be built with a different structure and the entire process repeated. Depending on the complexity of the network being modelled, it could take several hours on a single user Sun workstation to execute a few thousand training cycles. This makes the trial-and-error process very time consuming. The works discussed below required between 20,000 and 300,000 cycles to converge, once the correct structure was found. Finding the correct network structure in these environments was a non-trivial task!

However, there are two strong advantages to neural networks which make their use attractive. Firstly, it was much easier to impart existing information about a domain into a neural network, as this was done by showing examples of cause (inputs) and effect (outputs), then it was to explain every step in the logic between cause and effect. This was true for system control applications as it was for financial credit control applications. Secondly, the output from a neural network was not simply binary, but "confidence levels" or "degrees of certainty" in a particular result. When properly built and trained, a neural network presented with previously unknown inputs still generated a reasonable output, albeit with a lower level of confidence. This "fail safe" feature was quite an attractive one, as most conventional algorithms failed totally when a scenario was encountered which was outside the scope of the initial assumptions. Experience with the large body of work done in Image Processing [KLI90] and Optical Character Recognition has shown that these "guesses" are usually fairly accurate.

It was only recently that neural networks caused such an interest in the system control field. The works reviewed in that field fall into two main categories.

2.6.1. Conventional Back Propagation Neural Networks

In simpler control scenarios, the inputs to the network were the current status of the relevant parts of the domain, and the output from the network was the new control instruction required for the environment to remain stable. This was usually present in obstacle avoidance problems and some of the relevant work in that area warrants a quick description.

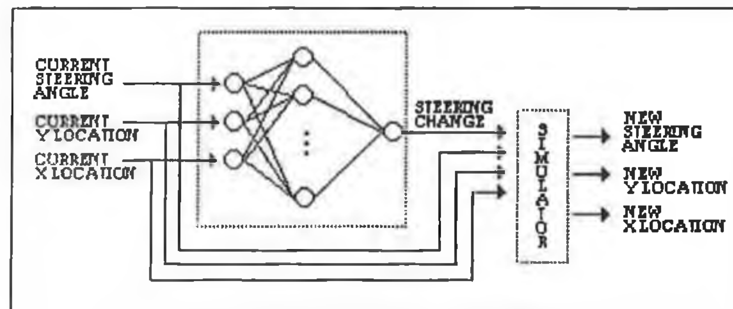
One collision avoidance project [OLI91] used the distance of the AGV from an obstacle and the angle to the obstacle relative to the AGV's heading as inputs to a neural network. Outputs from the network were the desired acceleration and change in heading for the AGV. This technique seemed to work well in the outdoor scenario used by the authors where the AGV had plenty of room for manoeuvring and with only stationary obstacles present in the domain.

These works demonstrated that dynamic control was possible using neural networks instead of using advanced modelling algorithms. In other words, these networks are capable of performing reactive tasks, such as process control [SAI91] [WIL91], path following [WAX87] [THO91] [TOU89] and collision avoidance successfully. However, note that in all of these works mentioned, there was no concept of time. They were not capable of planning a sequence of steps over a period of time and were therefore unsuitable for path planning.

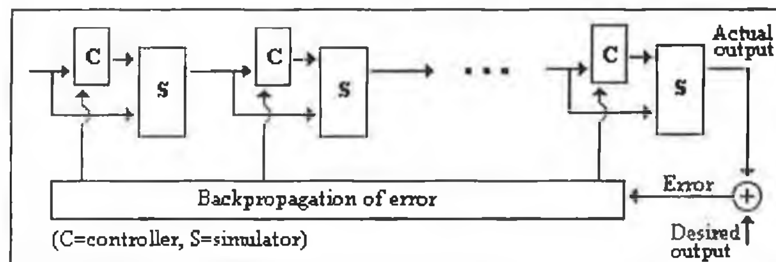
2.6.2. Back Propagation through Time Neural Networks

Some work was carried out on modelling time while using neural networks, which successfully found a path for a non-point AGV to the goal, while at the same time avoiding any stationary obstacles which might have been present in the domain.

In each of the following reports, the route to be navigated was determined (by the system designer) to require less than 'n' steps to complete (i.e. less than 'n' distinct movements by the AGV to reach the goal). The designer then builds a design of neural network which successfully performs collision avoidance for that particular AGV. A simulator was also found which would emulate the performance characteristics of the AGV, for the purposes of training the network. A total of 'n' neural networks and simulators are then assembled, so that the output from the first neural network fed directly into the first of the simulators. The output of the first simulator fed directly into the input layer of the second neural network, which output directly into the second simulator...



The intended output from the nth simulator was known - the AGV should be at a particular position (the goal), and usually at a particular angle. For a given starting position and angle of the AGV, it was AGV, it was possible to use a modelling algorithm to calculate the desired position and angle of the AGV after every distinct step. It was then simply a case of working forward, from a network to the subsequent simulator to the next network, until all 'n' steps have been reached. Any differences between the nth simulator's output and the desired final position of the AGV was viewed as an error. Standard Back Propagation algorithms were then used to propagate the error back through all 'n' networks. This completed one training cycle for the system of networks. As a result of this back propagation of the error, the next forward pass through the system of networks was less incorrect. Repeating this procedure with a

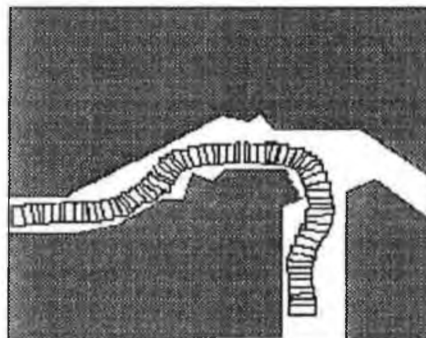


variety of different initial positions for the AGV, resulted in a network which was within tolerance levels of the results generated by the modelling algorithm. This training required a considerable amount of time, due to the error back propagation calculations and the number of training cycles required. Note however, that the time required for operating a trained neural network is minimal and certainly acceptable.

While all of the following reports implemented "Back Propagation through Time" using the sequences of neural networks outlined above, they have chosen different domain models, training sets and performance requirements.

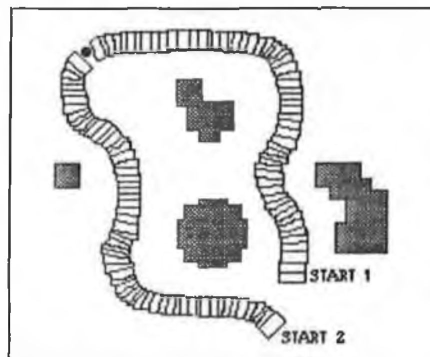
In [NGU89] and [NGU91], the authors trained a series of neural networks to navigate a jointed tractor-and-trailer AGV from an initial starting location and orientation to a goal location and orientation in a domain containing no obstacles. The authors reported that by changing the objective function (which generated the "correct answers" for the network to learn to imitate), they were able to force the AGV to minimise the length of the path travelled, or to minimise the number and extent of any turns required en route to the goal. It is interesting to note that the simulator which was used throughout this report was also a neural network - a trained reactive type of network - of the type outlined in 2.6.1 above.

This work was later extended [BIE92] to permit stationary obstacles in the domain. The authors reported that an AGV could be given a low level navigation instruction (i.e. turn right) and be able to amend the movement to suit the terrain. However, this was obstacle avoidance rather than path planning.



In [KOS92], the authors trained networks for navigating rectangular and tractor-and-trailer AGVs from an initial starting location and orientation to a goal location and orientation in a domain. No obstacles were present in the domain - either during training or during "live" runs. Unlike the work done in [NGU91] above, the simulators were implemented using simple kinematic equations, rather than neural networks.

In [PLU91], the authors trained a series of neural networks to navigate a jointed tractor-and-trailer AGV from an initial starting location and orientation to a goal location and orientation in a domain. During training, no obstacles were present in the domain. However, once the networks were



trained, stationary obstacles were added to the domain and the neural networks still navigated to the goal successfully - around obstacles if necessary. The author stated without proof that the techniques used could also cope with mobile obstacles in the domain. Note that the author's use of neural networks and potential fields was unique to all the reports in this area. It was the potential field aspect of this work which brought the authors to suppose that mobile obstacles could also be supported. The authors note the difference between their work, which avoided obstacles for 'n' steps while attempting to reduce the distance to the goal and a general purpose path planner.

2.7. Conclusion

There were several different approaches to path planning reviewed in this chapter. Their suitability, or lack thereof, is summarised below.

The Hightower and Hierarchical approaches did not guarantee to find a path to the goal in a domain with stationary obstacles and were excluded from further scrutiny.

There were two main weaknesses in applying the Map Sector approach to robotic navigation. Firstly, potentially valid paths could be blocked off because of how the sectors were marked. The Hierarchical Strategy went some way towards solving this problem, but did not guarantee to find a path, even though one might have existed. Secondly, all the various Map Sectoring approaches assumed only stationary obstacles were present in the domain. Extensions to allow mobile obstacles in the domain were either problematic (i.e. Hierarchical) or very slow (i.e. complete rebuilds of the domain at every time slice).

Due to similarities between the Landmark Navigation techniques and the manner in which humans navigate in the world, these techniques seem intuitively 'right' and appealing. However, initial landmark acquisition is non-trivial and very slow. Even with only stationary obstacles, simple two dimensional scenes took several CPU seconds to acquire and navigate on a single user DEC 2060 minicomputer. A simulated three dimensional version of this was developed on a Symbolics workstation and took considerably longer to run. Support for mobile obstacles (and possibly mobile landmarks) was problematic. Attempts to recognise which landmarks had moved and to only update those landmarks adjacent to the moved landmark required prior knowledge of all mobile obstacles in the domain, which was not a valid assumption for this project. The alternative, as with the Map Sector approach, was to update the entire list of landmarks in the domain at every time slice, which was computationally very expensive. These techniques were therefore deemed unsuitable and dropped from further consideration.

The Configuration Space technique successfully dealt with mobile obstacles and could even rotate the robot to fit through gaps if required. However, this technique could only achieve low level goals, such as obstacle avoidance and not higher level goals such as path planning, which was the focus of this investigation. It was therefore excluded from further study.

The main strong point in favour of using neural networks was that they would continue to function, and usually quite well, even in domains which are totally unknown to them. They are therefore more reliable and better than any standard algorithmic approach, which can simply fall down in unknown domains. However, the standard back propagation systems perform robust obstacle avoidance rather than path planning. The more complicated back propagation-through-time systems merely provide more sophisticated obstacle avoidance techniques, but still do not provide path planning functionality.

Visibility Graphs and Voronai Diagrams both represented the known domain as nodes in a network, with valid paths being edges between these nodes. They also stored the obstacles in their exact shape and the path was guaranteed to be found if one existed. However, the lack of ability to deal with mobile obstacles was important. While there were techniques for adding collision avoidance functionality to this approach, they increased the possibility for some unanticipated scenario to cause the entire system to fall apart and were therefore undesirable. We therefore felt more comfortable extending well known and reliable graph theory techniques to add mobile obstacle functionality, rather than using a new technique strapped onto graph theory. This was principally due to the provable stability of graph theory, which we found appealing. Also, a partial updating extension to this algorithm should reduce the computation times enough to support mobile obstacles in the domain.

Implementation details and experimental results for work on this partial updating extension are given in subsequent chapters. Both Visibility Graphs and Voronai Diagrams are suitable as a base for this extension, but as Visibility Graphs are significantly quicker to build then Voronai Diagrams, we based our work on Visibility Graphs for performance reasons.

3. Implementations based on Lee's Algorithm

During the course of our work, two variants of Lee's Algorithm were implemented and performance tests performed. This chapter outlines the design and implementation issues which were addressed. For benchmarks and analysis of these benchmarks, the reader is referred to Chapter 5.

3.1. Testbed Implementation Details

As the same basic functionality was required in all of the following experiments with Lee's Algorithm, a testbed implementation of Lee's Algorithm was developed. This one program contained all the common functionality that would be required in all of the following different versions of Lee's which were investigated during the course of this paper.

Developing the one testbed program for all the Lee's Algorithm experiments had significant benefits. Firstly, because of how this testbed program was structured, there was only one function which needed to be re-written when implementing any of the subsequent tests. This reduced the overall development time considerably. Secondly, as most of the code was exactly the same on a line by line basis, between one experiment and another, this meant that any performance differences in the experiments were totally and solely related to the function specifically implementing the different search techniques.

The basic structure of these testbed programs was as follows:

(i) Build a model of the domain, along with all mobile obstacles.

(ii) Update any moving obstacles and generate a pseudo "camera image" of the domain.

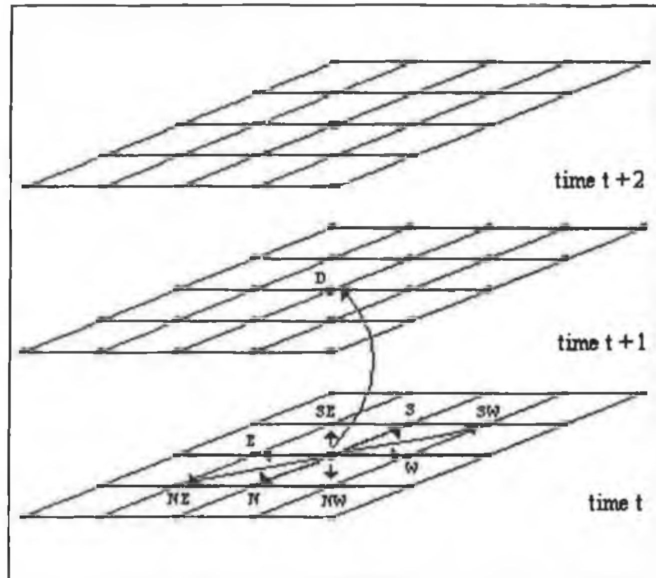
(iii) Use several different path planning techniques to find a suitable path from the AGV to the goal. Use the first step in this path as the movement command for the AGV.

(iv) Move the AGV. If the AGV is now at the goal location, then stop. Otherwise, loop back to (ii).

3.1.1. Representing the Domain

As was explained in Chapter 2 above, a fundamental assumption in Lee's algorithm was that all obstacles in the domain were stationary. As mobile obstacles must be supported in our experiments, some modifications had to be made in how the domain was represented to support this requirement.

By viewing the domain in a particular time-slice as a two dimensional array of points, it was possible to model the domain over a period of time by using a three dimensional array, where the x and y co-ordinates were two dimensions and time was the third.



Using this representation, stationary obstacles were modelled by being placed in the same x and y co-ordinates for all time-slices. Mobile obstacles, however, were modelled by changing their location in each time-slice. Note that this method of storing time as another dimension was similar in intent to the Hierarchical approach, but quite different in implementation and reliability. While the Hierarchical approach did not guarantee finding a path to the goal, modelling motion as a series of stationary obstacles in different positions during different time-slices allowed us to use the standard, well proven, working mechanism behind Lee's algorithm which guaranteed finding a path to the goal if one existed.

Every element in this three dimensional model represented a point in the domain. Even the current locations of the goal and the AGV were stored as elements in this array. Because of this, how information was stored in the array was important to the functionality of this testbed program. Every element in this array contained all the information which there was to know about that point in the domain. This included domain related information (i.e. was the point traversable or not, and how difficult or "expensive" was it to travel to it's adjacent nodes), as well algorithm related information (i.e. whether or not this point has been visited yet by the search algorithm, and if so, from which direction and what was the cost incurred in reaching this point from the current position of the AGV). As the status of a point changed (i.e. from traversable to non-traversable as an obstacle moved through the domain, or from unvisited to visited as the search progressed), these points in the array were updated to reflect their actual current status. A structure was used for storing all this information as follows:

```
typedef struct
	{
	float dist;
	int from;
	char type;
	float cost[9];
	} POINT;
```

`dist` kept track of the total distance travelled from the AGV to reach this point in the domain. Once we reach the goal, this value would be the cost of travelling from the AGV to the goal (i.e. the length of the path). This field was also used to trim down the amount of searching done by the algorithm. If we visited a point which has already been visited before, we compared the distance taken by the current visit against the distance taken by the previous visit. If the current distance was shorter than the previous distance, then the data stored in the point about the previous visit was overwritten by data about the current visit - as if it had never been visited at all. However, if the current distance was larger than the previous, then the current visit was simply abandoned. This procedure reduced the number of wasteful searches performed by the algorithm.

For visited nodes, the `from` field stored which direction the point was visited from. This was a simple Dynamic Programming technique which allowed us retrace our steps back from the current point to the AGV. Once the goal was discovered, this allowed us to retrace the entire path back to the AGV, thus giving the desired path to the goal.

The `type` field stored whether the point was traversable or not, as well as whether the point contained the AGV or the goal. Traversable and non-traversable points may change type as mobile obstacles move in the domain, but the stationary goal cannot be overrun by a mobile obstacle, nor can the initial location for the AGV.

The cost of travelling from one point to each of the adjacent points was stored in the array of `cost[9]`. These costs were initialised at the start of execution of the testbed program, and did not change. Note that there are 9 costs (and therefore 9 directions). This is the 8 horizontal directions (N, NE, E, SE, S, SW, W, NW), as well as the choice of not moving during a particular time-slice. Non-movement of the AGV was represented in this program by staying in the same (x, y) location and moving up from the array at time t to the array at time t+1.

3.1.2. Keeping track of points to visit

The basic Lee's algorithm explored a domain by searching in ever expanding circles from the starting point - in our case, the AGV. These circles continued to expand until either an obstacle was encountered or the goal was reached. At any given time, the set of points which make up the fringe of this ripple were being examined. If an obstacle was encountered, then the points where the obstacle was encountered were treated as other starting points, and the algorithm would start searching from there also. Part of this new "ripple" would overlap with the previous ripple, and the parts of the same area in the domain would be searched again.

This was obviously very wasteful in terms of computation time, and so Lee's algorithm contained a simple test which minimised this problem. If a point was visited which had already been visited before, we compared the distance taken by the current visit against the distance taken by the previous visit. If the current distance was shorter than the previous distance, then we overwrote the data stored in the point about the previous visit with data about the current visit. Later in the algorithm, the points adjacent to this point would be visited. However, if the current distance was longer than the previous visit, then further searching would be a waste of resources and the current visit was simply abandoned - as if the search had encountered an obstacle.

The most flexible approach for determining which points were due to be visited was to store all the nodes which were on the fringe of the search "ripple" as nodes in a double linked list, called the "Fringe List".

Initially all nodes which were adjacent to the AGV were placed into the fringe list. The program then unlinked the first node from the list and visited it. If a node was successfully visited, all of it's adjacent nodes were added to the list. If it was an unsuccessful visit (i.e. an obstacle, or a point previously visited with a shorter distance), then no nodes were added to the list. Either way, the visited node had been removed from the front of the fringe list, and the new first node in the list was now examined. This procedure was repeated until the goal was reached.

In the case where there was no valid route from the AGV to the goal, all nodes in the fringe list will have been removed from the list, visited in turn and the goal still not found. In our implementation, if the list was empty (therefore no more points in the domain were eligible for a visit) and the goal had not been found, then no valid path to the goal existed. The testbed program checked for this and if necessary would advise the user and then terminate.

The information stored within the fringe list and the structure used is outlined below. The basic technique used for implementing the fringe list was a double linked list. Each node in the fringe list only contained the minimum of information:

```
typedef struct daisy
        {
        int w,l,h;
        float dist;
        struct daisy *prev,*next;
        } NODE;
```

`w,l,h` store the x, y, and time t dimensions of the co-ordinates for a point in the domain which will be visited.

`dist` is the total distance which will have been travelled to the point from the AGV's starting location. When a point is visited, it will be checked to see if it had been visited previously. If it was previously unvisited, this distance is stored in the `dist` field in the relevant array element. However, if the point had been previously visited, then this `dist` is compared with the `dist` in the array element. The shorter distance is stored in the element.

Remember that any other information which may be needed about this point in the domain can be found by examining the element in the array whose co-ordinates are [w][l][h].

3.2. Standard Lee's Algorithm

The algorithm started by inserting the point where the AGV was located into the empty fringe list. A node was then removed from the front of the fringe list and the point contained within that node made the current point. If the current point was not the goal and was successfully visited (i.e. was not an obstacle, was not already visited or was not already visited in a shorter distance) then all points adjacent to the current point were appended to the fringe list. This procedure was repeated until either the goal point was visited, and therefore the goal was found, or until there are no more points in the DLL, and therefore the path to the goal could not be found.

Note that by always inserting new nodes into the fringe list at the end of the list, this ensured that all other points already in the fringe list were visited before these newly added points. The effect of this was to force a breadth-first search of the domain.

3.3. Lee's Algorithm with Straight Line Bias

We noticed that no attempt was made in Lee's algorithm to utilise the known location of the goal. The following extension to Lee's algorithm utilised this knowledge to reduce the amount of searching required to find a path to the goal.

Note that, unlike the conventional Lee's Algorithm above, which always appended new nodes to the end of the fringe list, nodes were inserted into the list sorted by their distance from the goal. This distance was calculated using fundamental co-ordinate geometry. No attempt was made to detect obstacles which might be encountered en route - the result was simply the length of the straight line from the node to the goal. This simple technique for estimating the distance from the node to the goal was acceptable, because it was used to determine the *relative distance* from the node to the goal (relative to the other fringe

nodes). It did not attempt to calculate the actual remaining distance the AGV had to travel from that node to reach the goal.

4. Implementations based on Graph Theory

During the course of our work, several variants of graph theory were implemented and performance tests carried out. This chapter outlines the design and implementation issues which were addressed. Benchmarks, and analysis of these benchmarks, were provided in Chapter 5.

4.1. The Partial Update Technique Explained

We discussed in a previous chapter how conventional graph theory provided path planning functionality in a domain with no mobile obstacles. We also showed how the graph network being used to represent the domain could be completely rebuilt after each time-slice, thus allowing this technique to work with mobile obstacles in the domain. In order to improve performance of this work around, we developed an extension to conventional graph theory which we have outlined below. Implementation details of conventional graph theory were given in earlier chapters. Note that the terminology used throughout this chapter was consistent with that introduced in 2.3 above.

In 2.3 above, the feasibility of using graph theory for path planning in a domain with mobile obstacles was discussed. The main weakness with this approach was the lack of built-in support for mobile obstacles - they were all assumed to be stationary. Updating the entire graph network after every time-slice solved this problem, as it made all mobile obstacles appear as stationary obstacles in that time-slice. However, the overhead involved in rebuilding the entire network model of the domain after every time-slice was problematic. To reduce this overhead, we developed an extension that identified all the parts of the network which needed to be updated after each time-slice. Using this information, the unchanged parts of the network were left intact, and only the changed parts rebuilt, thus significantly reducing the time required to update the network.

It is worth pointing out that a partial update extension for a graph tree, as opposed to a graph network, is quite straightforward - the "parent" and the immediate descendant nodes need to be updated. The structure of the graph tree ensured that these were the only nodes directly effected. However, graph networks and their inherent interconnected design were more complicated.

Recall that the only input to the path planning software was a two dimensional array, which simulated a simplified camera image of the entire domain. When the new camera image was compared with the previous camera image it was possible to detect what parts of the domain had changed. From this information, we deduced which vertices no longer existed (and should be removed from the network) or needed to be created (and should be inserted into the network). Once this stage was completed, we had an

updated set of nodes in the graph network. However, the edges between these nodes were only partially updated, and did not correctly represent the true state of the domain.

The simplest solution which would keep all the edges in the network up to date was to destroy all edges on all nodes throughout the entire network, and then rebuild them. While this would guarantee to keep the edges in the network up to date, it was also too time consuming for our requirements. Ideally, we only want to update the edges which need to be updated. A solution was proposed as follows:

(i) A list, called the update list, was created and initialised at the end of each time-slice. Any node which changed in any way at the end of a time-slice would be added to this list. All nodes in this list would be scrutinised and rebuilt as necessary. All nodes not in this list would not be altered or examined in any way.

(ii) The current camera image of the domain in the current timeslice was compared with the camera image from the previous timeslice. Only the parts of the current image which differed with the previous image needed to be checked for vertices. New nodes may need to be inserted into the graph. Also, some nodes may no longer be valid vertices and must therefore be removed from the graph. When all the relevant nodes were updated, the edges between these nodes needed to be rechecked.

(iii) Before a node was deleted from the network, all nodes adjacent to it were added to the update list. As each node was added to the update list, all edges from that node to other nodes in the domain were deleted. Only then would the original node be deleted. This procedure was repeated for all nodes which needed to be deleted in the network. It was possible, and indeed likely, that if more than one node was deleted in the domain, they would have shared a common adjacent node. Attempts to insert a node into the update list more then once were simply ignored.

(iv) When a new node was inserted into the network, it would be added to the update list. All adjacent nodes to this new node were then calculated and also added to the update list. As each node was created and then added to the update list, it's list of edges was initialised as being empty. This procedure was repeated for all nodes inserted into the network.

(v) A node in the update list would be made the current node. All nodes in the network which were adjacent to the current node were then found and an edge built between them and the current node. Finally, the current node was removed from the update list. This procedure was then repeated for every node in the update list, until the list was emptied.

(vi) The network was now updated and represented the domain in the next time-slice. Once the entire graph network for this time-slice was generated, the camera image of the domain in the current time-slice was preserved for comparison with the camera image in the next time-slice.

Note that the nodes moved into the update list had all their existing edges deleted and later all edges rebuilt, even though this might result in some edges being deleted and then immediately rebuilt. This was done because the benefits of the simpler implementation outweighed the small potential performance improvement which might or might not emerge depending on the node in question.

Using this extension, we intended to update the network of nodes and edges much quicker than simply updating the entire network - regardless of which graph network searching algorithms were used. Implementation details were discussed later in this chapter. Source code and comparative benchmarks for both the conventional graph theory approach and also this extension were presented and discussed in later chapters.

4.2. Testbed Implementation Details

As the same basic functionality was required in all of the following experiments with Graph Theory, a testbed implementation of a Graph Theory program was developed. This one program contained all the common functionality that would be required in all of the following different versions of Graph Theory which were investigated during the course of this paper.

Developing the one testbed program for all the Graph Theory experiments had significant benefits. Firstly, because of how this testbed program was structured, there was only one function which needed to be re-written when implementing any of the subsequent tests. This reduced the overall development time considerably. Secondly, as most of the code was exactly the same on a line by line basis, between one experiment and another, this meant that any performance differences in the experiments were totally and solely related to the function specifically implementing the different search techniques.

The basic structure of these testbed programs was as follows:

(i) Build a model of the domain, along with all mobile obstacles.

(ii) Update any moving obstacles and generate a pseudo "camera image" of the domain.

(iii) Build a graph network which accurately represents the domain portrayed in the camera image.

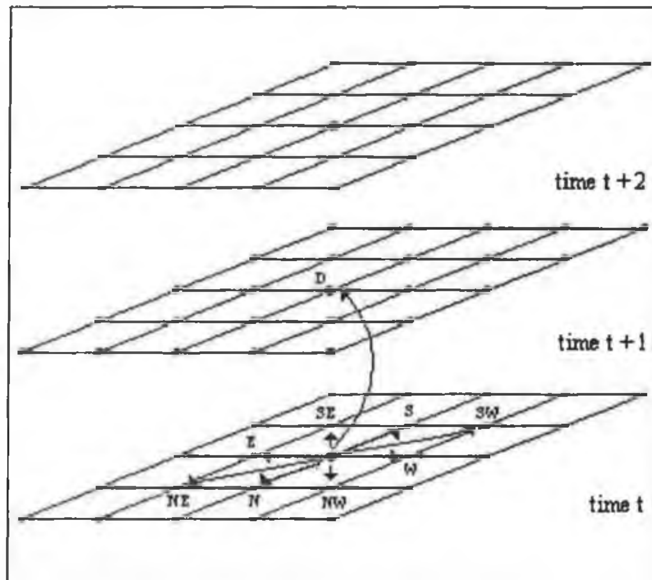
(iv) Use several different path planning techniques to find a suitable path from the AGV to the goal. Use the first step in this path as the movement command for the AGV.

(v) Move the AGV. If the AGV is now at the goal location, then stop. Otherwise, loop back to (ii).

4.2.1. How the Domain was Represented

As was explained in Chapter 2 above, a fundamental assumption of graph theory was that all obstacles in the domain were stationary. As mobile obstacles must be supported in our experiments, some modifications had to be made in how the domain was represented to support this requirement.

By viewing the domain in a particular time-slice as a two dimensional array of points, it was possible to model the domain over a period of time by using a three dimensional array, where the x and y co-ordinates were two dimensions and time was the third.



Using this representation, stationary obstacles were modelled by being placed in the same x and y co-ordinates for all time-slices. Mobile obstacles, however, were modelled by changing their location in each time-slice.

Every element in this three dimensional model represented a point in the domain at a given time-slice. Even the current locations of the goal and the AGV were stored as elements in this array. Because of this, how information was stored in the array was important to the functionality of this testbed program. Every element in this array contained all the information which there was to know about that point in the domain. This included domain related information (i.e. was the point traversable or not, and how difficult or "expensive" was it to travel to it's adjacent nodes), as well algorithm related information (i.e. whether or not this point has been visited yet by the search algorithm, and if so, from which direction and what was the cost incurred in reaching this point from the current position of the AGV). As the status of a point changed (i.e. from traversable to non-traversable as an obstacle moved through the domain, or from unvisited to visited as the search progressed), these points in the array were updated to reflect their actual

current status. The following structure was used in C++ for storing all this information in a Domain class:

```
typedef struct
	{
	float dist;
	int from;
	char type;
	float cost[9];
	} POINT;
```

dist kept track of the total distance travelled from the AGV to reach this point in the domain. Once we reach the goal, this value would be the cost of travelling from the AGV to the goal (i.e. the length of the path). This field was also used to trim down the amount of searching done by the algorithm. If we visited a point which has already been visited before, we compared the distance taken by the current visit against the distance taken by the previous visit. If the current distance was shorter than the previous distance, then the data stored in the point about the previous visit was overwritten by data about the current visit - as if it had never been visited at all. However, if the current distance was larger than the previous, then the current visit was simply abandoned. This procedure reduced the number of wasteful searches performed by the algorithm.

For visited nodes, the **from** field stored which direction the point was visited from. This was a simple Dynamic Programming technique which allowed us retrace our steps back from the current point to the AGV. Once the goal was discovered, this allowed us to retrace the entire path back to the AGV, thus giving the desired path to the goal.

The **type** field stored whether the point was traversable or not, as well as whether the point contained the AGV or the goal. Traversable and non-traversable points may change type as mobile obstacles move in the domain, but the stationary goal cannot be overrun by a mobile obstacle, nor can the initial location for the AGV.

The cost of travelling from one point to each of the adjacent points was stored in the array of **cost[9]**. These costs were initialised at the start of execution of the testbed program, and did not change. Note that there are 9 costs (and therefore 9 directions). This is the 8 horizontal directions (N, NE, E, SE, S, SW, W, NW), as well as the choice of not moving during a particular time-slice. Non-movement of the AGV was represented in this program by staying in the same (x, y) location and moving up from the array at time t to the array at time t+1.

4.2.2. The Camera Image Generating Module

The purpose of this module was to simulate a "bird's eye" camera view of the entire domain. There were two distinct parts to this module.

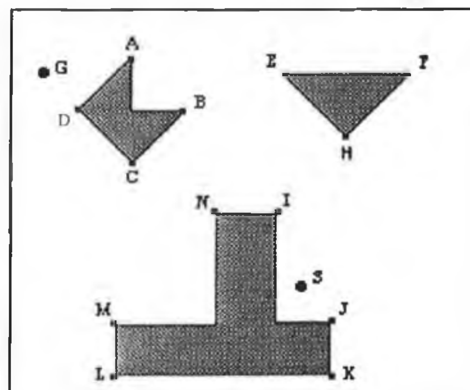
(i) A model of the domain. The implementation details for this were outlined in 4.2.1 above.

(ii) A double link list, called the obstacle list, was built of all the mobile obstacles in the domain. This stored information which was unique to each mobile obstacle, such as it's shape, velocity and direction. The shape of every obstacle was specified at the start of the program, and each obstacle was of a different shape and size. The initial direction and initial location for each object was chosen on a random basis. Each obstacle has a maximum velocity which was slightly less than that of the AGV.

Every time this module was called to generate a camera image of the domain, it updated each of the mobile obstacles in the obstacle list using the information available on each mobile object, as well as a knowledge of the existing static obstacles in the domain. If moving a mobile obstacle in the current direction would cause a collision to occur, the given direction for the mobile obstacle was changed to a random new direction. To simulate turning time for the mobile obstacle, the obstacle waited one time-slice before starting to move in the new direction. As each of these mobile obstacles was updated, it's representation in the model of the domain was also updated.

When all of these mobile obstacles in the obstacle list were updated, the model of the domain reflected the status of the domain in the current time slice. An image of this updated domain model was then generated and passed to the image processing module.

4.2.3. How a Graph Network was Generated from a Camera Image



The only input to this module was the image generated by the "camera simulator" above.

This module searched for all prominent points on all obstacles in the supplied camera image. This set of prominent points became the nodes or vertices in the visibility graph for that domain. A handy way of describing what were the valid vertices for an obstacle was to say that

to say that by connecting all of an obstacle's vertices together, the obstacle should be entirely enclosed. The goal and AGV were represented as point sized obstacles in the image. Note that, in this technique, all obstacles were considered to be stationary - at least for the time-slice when the graph network was being generated. Because of this, all obstacles, both stationary and mobile, in the domain were represented in the same manner in the camera image. A different graph network would be generated in the next time-slice, with obstacles in different locations, but it was still considered to be dealing with stationary obstacles.

When all the vertices were generated, the next step was to determine which nodes were adjacent to which. This was done by attempting to draw straight lines from every vertex to every other vertex in the graph. If no obstacle was encountered between two vertices then those two vertices were declared to be "adjacent" to each other and a common edge established.

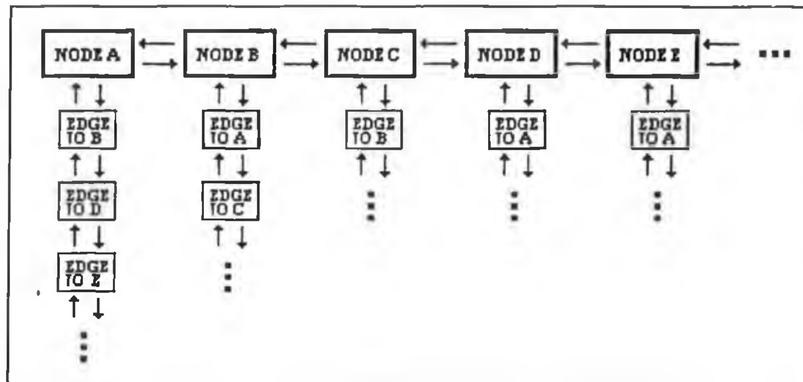
After every time-slice, a new image was presented to this module, and the entire graph rebuilt. As the time required to totally rebuild a network was too long for our purposes, we developed a technique for only detecting which parts of the network required updating, and leaving the other parts unchanged. A detailed explanation and implementation can be found in 4.1 above.

4.2.4. How the Graph Network was Represented

Conventional graph theory represented this information in a symmetrical two dimensional array called an "Adjacency Matrix" - so called because it stored which nodes were adjacent to which. Every vertex in the domain was represented by an element in this array. If it was possible to travel from node 'a' to node 'b', then nodes 'a' and 'b' were said to be adjacent. and the element [a][b] in the matrix was set to TRUE. If 'a' and 'b' were not adjacent, then it was set to FALSE.

However, there was a problem with this approach. As obstacles moved within the domain, the number of vertices changed constantly. This entailed adding or deleting nodes from the graph network, or to be more precise, changing the number of elements in the adjacency matrix. Implementing this was very problematic. The solution was therefore not to implement the adjacency matrix as an array. A more flexible alternative would be to use a technique called "Adjacency Lists". While adjacency lists were more flexible, and allowed the addition and deletion of nodes in the network throughout the operation of the program, they were significantly more complicated to implement. Part of the adjacency list for the same diagram was given as an example.

All vertices in the domain, including the AGV and goal locations, were represented by nodes in a double link list called the vertex list. Each of the nodes in this vertex list contained a pointer to the start of another double



another double link list - a list for edge nodes. The edge list contained a node for every edge to an adjacent node in the graph network. The structures used for implementing these lists in C++ were:

```
typedef struct vn
        {
        int t,w,l;
        char nodeType;
        struct vn *searchPrev, *searchNext;
        float searchDist;
        int searchMarker;
        EdgeList *edgeList;
        struct vn *pathFrom, *pathTo;
        struct vn *prev, *next;
        } VERTEX_NODE;
```

for the vertex nodes in the VertexList class, and

```
typedef struct van
   {
        int t,w,l;
        float dist;
        struct van *prev, *next;
        } EDGE_NODE;
```

for the edge nodes in the EdgeList class.

4.2.5. How the Graph Network was Updated After Every Timeslice

At the end of every timeslice, any movement in the domain renders the graph network out of synch with the actual domain. The graph network must therefore be updated after every timeslice.

The conventional (and certainly simplest) technique for this was to completely destroy the entire graph network and to then completely rebuild a new graph network which did accurately represent the domain. Obviously, in domains where some vertices remain unchanged from one timeslice to another, some room existed for optimisation.

The Partial Updating technique developed elsewhere in this paper was one attempt at optimising this process. In order to test the effectiveness of this technique, benchmarks were carried out as part of this paper using both of these techniques and the results analysed.

4.2.6. Keeping track of nodes to visit

Given that the adjacency lists contained an up to date set of nodes and edges in the entire domain, there were several different techniques available for planning a path through the network which this represented. However, all of these path planning techniques needed some method of remembering which nodes in the network had already been examined, which were currently being examined, and therefore which were to be examined next. This was implemented using yet another double link list called a Fringe List and a step counter, which kept track of the current time slice, or step, throughout the entire experiment.

The Fringe List was so called because all the nodes in this list were on the fringe of the search space. Initially all nodes which were adjacent to the AGV were placed into the fringe list. The program then unlinked a node from the front of this list and visited it. If it was an unsuccessful visit (i.e. the node turned out to be an obstacle, or a point previously visited with a shorter distance), then the nodes adjacent to the visited node were not added to the list. However, when a node was successfully visited, all of it's adjacent nodes were added to the fringe list. Either way, the visited node was removed from the front of the fringe list. The new front node in the list was then examined and this procedure repeated until the goal was reached, or the fringe list was empty.

In the case where there was no valid route from the AGV to the goal, all nodes in the fringe list will have been removed from the list, visited in turn and the goal still not found. In our implementation, if the list was empty (therefore no more points in the domain were eligible for a visit) and the goal had not been found, then no valid path to the goal existed. The testbed program checked for this and if necessary would advise the user and then terminate. If the goal was encountered before the fringe list was empty, then a valid path to the goal obviously existed and had been found.

All of the different search techniques used in this paper utilised this technique. The main difference between the different search techniques was the sorting order used for the nodes added into the fringe list. Breadth First searching techniques always added new nodes to the end of the list, Depth First added to the start of the list, while Priority First and A* had the nodes in a sorted order, with the most important nodes placed nearest the front of the list.

As a node was visited, it was marked not with a simple TRUE/FALSE Boolean flag, but with an integer representing the current number of cycles since the experiment started. This would store the cycle in which the node was visited. A step counter was used to keep track of the current cycle, and this was

incremented every time the AGV moved, or even when the AGV waited in the same position, and advanced a time-slice. To check if a node had already been visited, the value of the step counter was compared with that in the node. If they matched, the node had already been visited *during the current cycle*. If they did not match, the node had not been visited *during the current cycle* and was treated as an unvisited node.

This step counter idea was used in all implementations of the different graph searching techniques in this paper. It's main advantage was that it removed the need to re-initialise every node in the graph as being "unvisited" at the end of every cycle. Simply resetting the start of the Fringe List to empty was now sufficient at the end of a cycle, with obvious performance benefits.

4.3. Graph Search Algorithms Used

There were four different graph searching techniques used with both methods of updating. For a proper description of the Breadth First, Depth First, Priority First and A* search algorithms, the reader is referred to Chapter 2.

In implementation, the one fundamental testbed program was used for all four search algorithms. The were only two differences between the eight programs tested.

(i) The position in which nodes were inserted into the fringe list varied depending on the search technique being used. Breadth First always appended new nodes at the end of the list, Depth First always inserted at the beginning of the list, while Priority First and A* maintained the list sorted by the nodes proximity to the goal.

(ii) Depending on the graph updating technique being used, the graph was either completely rebuilt or only partially updated at the end of every time-slice.

As the rest of the programs were identical, on a line for line basis, any differences in performance were solely related to the various algorithms tested. The results and their analysis are presented in the following chapter.

5. Benchmarks and Performance Analysis

While the theoretical benefits and drawbacks for each of the path planning techniques described so far have been noted, there was no replacement for live testing or at least realistic simulation. With this in mind, a range of tests were developed to judge the effectiveness of each of the technique outlined above. The various steps involved in the benchmarking phase of this project were detailed in this chapter.

5.1. Test Data Used

As was mentioned briefly in 1.5, we were interested in testing the performance of the different navigation algorithms in three models. These models, and our interest in them, were as follows:

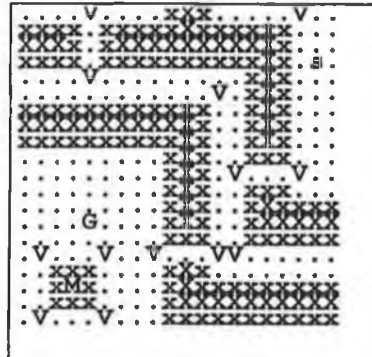
(i) Single AGV with Stationary Obsticles. This model contained only one AGV and several static obstacles. At no stage could were there any mobile obstacles present. The AGV attempted to travel from it's starting location to it's intended goal, without colliding with any obstacles en route. This model represented automated warehouses, where an AGV was used to store and retrieve goods from storage areas and humans only entered for occasional maintenance checks when the AGV was stopped.

(ii) Single AGV with Mobile Obstacles. This model contained only one AGV and a mix of both static and mobile obstacles. The AGV attempted to travel from it's starting location to it's intended goal, without colliding with any of the obstacles en route. The mobile obstacles constantly travelled through the domain, and it was the responsibility of the AGV to detect a pending collision and take steps to avoid them. This model represented an automated warehouse like (i) above which also permitted humans to safely travel through the warehouse while the AGV was in operation.

(iii) Multiple AGVs. This model contained multiple AGVs, and several static obstacles. Each AGV had it's own intended destination to reach, and treated any other AGVs which were encountered en route in exactly the same manner as any other mobile obstacle. This model represented an automated warehouse like (i) above which permitted both humans and other AGVs to safely travel through the warehouse without collision.

For each of these three models, there were several sample domains created which each contained a different starting position for the AGV, goal and any mobile obsticles which might be present. The domains were developed to judge the effectiveness of the path planning techniques outlined in this paper, when used in each of the three models detailed above.

Each domain measured 20 units by 20 units in size, and allowed up to 10 consecutive timeslices to be modelled at the same time. Within this domain, there were some stationary obsticles present, which represented walls and doorways, denoted by an uppercase 'X'. There was an AGV, or starting position for the search, denoted by an 'S'. There was also a goal position, denoted by a 'G'. Finally, there could be one or more mobile obsticles present in the domain - these were each denoted by an 'M'. All obsticles were surrounded by a safety zone, denoted by a lowercase 'x'. At every extremities of each obsticle was a vertex, denoted by a 'V'. This simple two dimensional pattern was stored for time t, t+1, t+2, t+3 and so on up to t+10.



For each AGV present in the model, ten different starting and ending locations were picked at random within the domain and stored in a data file. From zero to three mobile obstacles were also present in the domain and their locations were also picked at random and stored in the same file.

All of these mobile obstacles moved in totally random directions, but travelled in straight lines until they were blocked by an obstacle. A different direction was then selected at random and the obstacle kept moving. This behaviour pattern was chosen in preference to the "Brownian Motion" or Chaos Theory models, where all mobile obstacles travelled in apparently random directions and change direction frequently. These models were better suited to domains more congested then those modelled in this paper.

Obviously, as each technique could plan different routes to the goal, and also as mobile obsticles changed direction randomly after a collision, the set of free spaces available in the domain could be radically different within a few execution steps. This could not be made consistent between each experiment.

Each of the domains was intended to be used as a testbed for all of the different navigation techniques which were under scrutiny in this paper. To ensure that the results could be accurately compared, a given domain had to be identical while being tested by each of the navigation techniques. As the obstacles moved in a random direction, it was possible that any obstacle could randomly move in the path of the robot, causing it to stop or take evasive action before continuing on towards the goal. The extent to which this happened varied between trial runs. To smooth out the impact that this had on the final result, each of the experiments were carried out 10 times and the results averaged. This minimised the impact of the random movements and allowed for easier assessment of the effectiveness of the various techniques in these scenarios. It also minimised any impact that hardware or operating system related events would impinge on the results.

For each time-slice, the location of the robot and of all obstacles in the domain were noted and the cheapest path through the environment found. While the actual domain was a dynamic one, with mobile obstacles and a mobile robot, it was represented as an individual sequence of static time-slices, where each time-slice was treated as an distinct static domain. While the techniques used found the cheapest path through the static time-slices, this did not guarantee that the complete path travelled by the robot to the goal was the absolute cheapest possible path to that goal. However, this limitation was acceptable.

5.2. Test Results Recorded

The following values were recorded for every trial run carried out:

(i) The Elapsed Time recorded how long the test program took to run to completion. This excluded the time required to load configuration files and initialise the relevant data structures required by the algorithm on start-up or to save the results to disk on completion. However, everything else - including screen activity - was included. All time was measured in milliseconds.

(ii) The Compute Time was the actual time taken for the technique to search it's internal representation of the domain for a path to the goal. Nothing else was included in this timing, not even the time required to display the domain and any progress information on the screen. Again, all the units used were milliseconds.

(iii) The Distance was the total distance travelled by the robot from the starting location to the destination location, along the path determined by the current algorithm. The distance units used were not specified. This allowed the reader to use whatever units were relevant to scale both the distance travelled and also the size of the domain within which the robot was operating.

(iv) The Number of Cycles was the number of complete iterations of the find path, move robot, move objects loop required for the robot to actually reach the destination.

(v) The Compute Time per Cycle was calculated using the recorded values for the Compute Time for the entire experiment (see (ii) above) and the Number of Cycles required for the entire experiment (see (iv) above). This figure showed the average amount of compute time needed per cycle during the experiment. This value was used to measure the efficiency and relative performance of the different techniques under scrutiny. All units used were in milliseconds.

5.3. Analysis of Benchmarks

The first conclusion from this work was that the same basic path was found by all of the different algorithms benchmarked. The only variations were caused by the AGV deviating from it's original path to avoid colliding with a mobile obstacle.

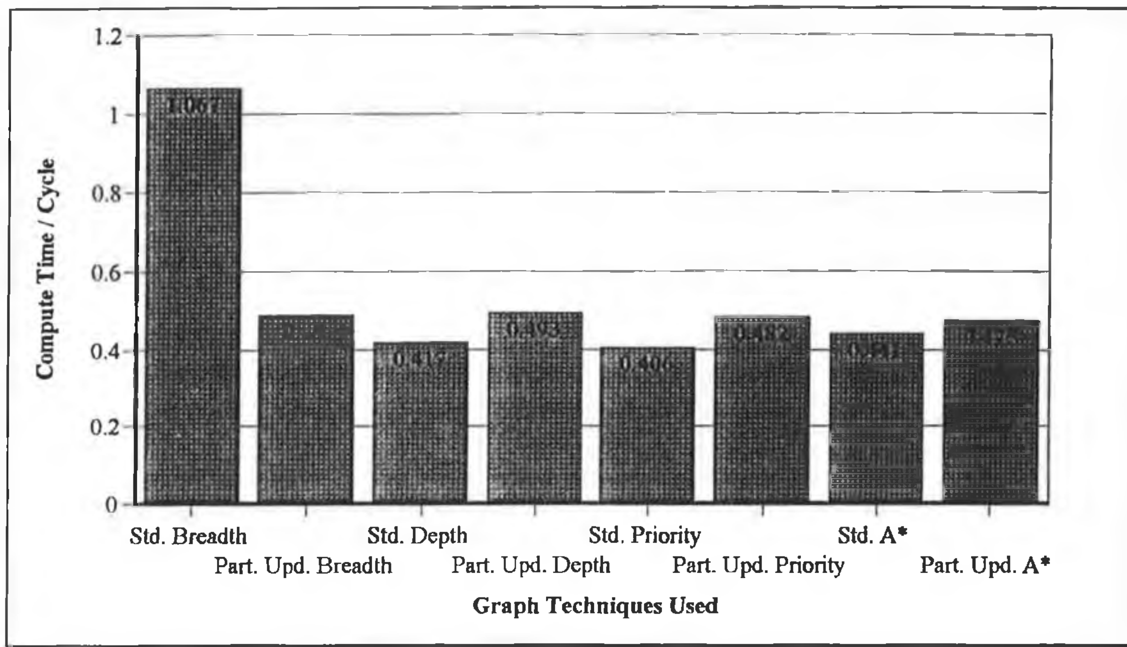
However, the calculation time required to find that path varied considerably. Both the Standard Lee's Algorithm and it's variation, developed earlier in this paper, were several orders of magnitude slower then even the slowest of the Graph Theory techniques. Both Lee's Algorithms investigated in the course of this paper were therefore dismissed as being far too slow to be of further interest and were excluded from the remainder of this chapter for the sake of clarity.

In the rest of this chapter, results were presented which showed the performance of the different graph search techniques in conjunction with both the Standard and Partial Updating techniques in the different test scenarios.

5.3.1. Single AGV in a Static Domain

The first model we investigated contained only one AGV travelling in a domain which contained only stationary obstacles, such as walls and corridors. This domain attempted to model how a single AGV and it's navigation software coped in a static environment such as the most simplistic of automated warehouse, where no human access was permitted while the solitary AGV was in operation.

Ten different scenarios which fitted the requirements for this model were created and then tested using each of the different graph searching techniques. The performances of these different techniques were collated and summarised in the following graph:



As explained above, the average compute time per cycle was used as the measure of performance. As can be clearly seen, the performance of the Partial Update Techniques was extremely consistent, regardless of which search technique was used. By contrast, the performance of the Standard Update Techniques varied significantly depending on which search technique was used. Overall, the Standard Update Techniques needed an average Compute Time per Cycle of 0.583 ms. By comparison, the Partial Update Techniques needed an average Compute Time per Cycle of 0.485 ms - a performance improvement of approximately 17%. To reduce deviation in the averaged times, both the worst and best performaces were then excluded, to reveal that now the Standard Update Techniques reduced it's average Compute Time per Cycle to 0.429 ms while, by comparison, the Partial Update Techniques remained almost unchanged with an average Compute Time per Cycle of 0.486 ms. The performance improvement of approximately 17% suddenly became a performance drop of approximately 12%.

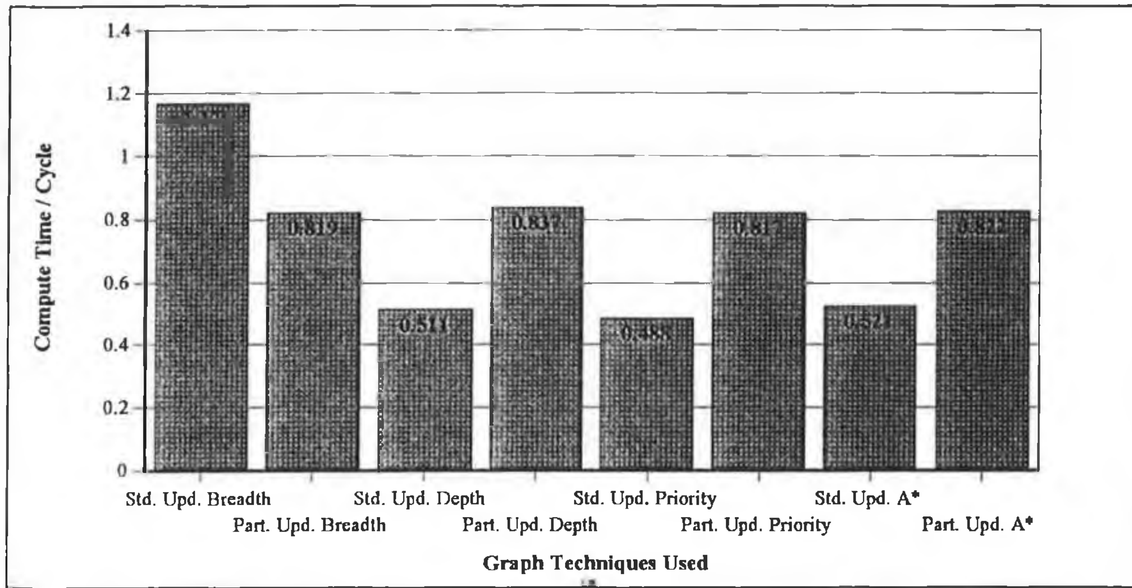
For a more detailed display of these findings, please note that the tables of actual results were enclosed in Appendix A, along with tables which showed the averaged data used to generate these graphs. Details on the features and assumptions used in these domains were given in Chapters 4.

5.3.2. Single AGV in a Domain with Independent Mobile Obstacles

The second model we investigated also contained static obstacles and only one AGV, like the model detailed above. However, it also contained a small number of completely independent mobile obstacles which moved through the domain totally oblivious to any attempt by the AGV to reach it's destination.

The intended paths for these mobile obstacles were not known to the AGV or it's navigation software. This model was used in an attempt to simulate a more flexible automated warehouse which allowed human access to the warehouse floor while the AGV present was still in operation. The AGV still had to reach it's destination without colliding with any humans who strayed into it's intended path.

In total, there were thirty different scenarios, which fitted the requirements for this model, created and tested. Ten scenarios contained just one mobile obstacle, another ten scenarios contained two independent mobile obstacles, and finally another ten scenarios contained three independent mobile obstacles. As with 5.3.1 above, each of the graph traversal algorithms were tested on all of these scenarios. As before, the results were summarised in the following graph:



As explained above, the average compute time per cycle was used as the measure of performance. As can be clearly seen, the performance of the Partial Update Techniques was extremely consistent, regardless of which search technique was used. By contrast, the performance of the Standard Update Techniques varied significantly depending on which search technique was used. Overall, the Standard Update Techniques needed an average Compute Time per Cycle of 0.673 ms. By comparison, the Partial Update Techniques needed an average Compute Time per Cycle of 0.824 ms - a performance downgrade of approximately 18%. To reduce deviation in the averaged times, both the worst and best performaces were then excluded, to reveal that now the Standard Update Techniques reduced it's average Compute Time per Cycle to 0.516 ms while, by comparison, the Partial Update Techniques remained almost unchanged with an average Compute Time per Cycle of 0.821 ms. The performance improvement of approximately 18% suddenly became a performance drop of approximately 37%.

For a more detailed display of these findings, please note that the tables of actual results were enclosed in Appendix A.2, along with tables which showed the averaged data used to generate these graphs. Tables and graphs were also supplied for each individual experiment with just one mobile obstacle, then just two mobiles and finally with just three mobiles present in the domain at the time. Details on the features and assumptions used in these domains were given in Chapters 4.

5.3.3. Multiple AGVs in the same Domain

An interesting side-effect of the work done in the above section with a single AGV in a domain with independent mobile obstacles quickly became apparent. The same techniques could also be used in a domain with more then one independent AGV active at a given time. This was possible because of how mobile obstacles were dealt with.

In the previous model, the AGV navigated through a domain to it's goal and avoided any mobile obstacles which were encountered en route. This was done without making any attempt to predict the intended path of these mobile obstacles, but simply allowing a safety zone around each obstacle. This same technique could easily be applied to a domain where multiple independent AGVs navigated to their respective goals, using the same navigation software, without colliding with each other or any other obstacles which might also be present in the domain.

As far as each individual AGV was concerned. it didn't matter if the mobile obstacles present in the domain were other independent AGVs, humans wandering through the domain or simply stationary obstacles. As each AGV ran it's own complete instance of the same navigation software, each AGV viewed all other AGVs as mobile obstacles from it's viewpoint, constructed safety zones and vertices to reflect this and navigated around them accordingly - using technology successfully proven in the previous section.

However, it is important to note that the navigation software treated the current AGV as the only AGV in the domain *at that moment* and all other AGVs as mobile obstacles. Once a single navigation instruction had been issued and executed for the current AGV, the next AGV in the domain was made the current AGV and the procedure repeated. Each of the AGVs worked totally independently of each other and at no stage did any AGV know the intended paths of any of the other AGVs or any mobile obstacles which might be in the domain.

6. Conclusion

In this paper, a number of different techniques were presented for navigating an AGV through a domain to it's goal. Of these, some were existing algorithms and others were extensions developed during the course of this paper. After an initial review of all the techniques, the more relevant techniques were implemented and their relative performances compared and evaluated. It should be pointed out from the outset that all of the various techniques which were implemented during the course of this paper did successfully find a path from the AGV to the goal in a domain with both stationary and mobile obstacles. Usually, the path was found by all, but any variations did occur, these were necessary to avoid mobile obstacles which had strayed into the AGV's original path. However, the relative performance of the different techniques was quite varied.

The overall performance of the standard Lee's Algorithm was by far the slowest of all the techniques examined in this paper, by at least several orders of magnitude. The Straight Line variant of this algorithm, which was developed in the course of this paper, proved to be consistently faster then the original, but was still orders of magnitude slower then the slowest Graph Theory techniques investigated. Both of these Lee's techniques were therefore deemed impractical and dropped from any further consideration.

Performance of the various Graph techniques implemented varied considerably, although they all proved to be considerably faster then either of the Lee's Algorithm techniques. The question as to which Graph technique returned the best performance really depended on the domain. In domains where there was only one AGV and no mobile obstacles at all, the Partial Update graphing techniques performed faster on average then the Standard Updating techniques. However, with one mobile obstacle in the domain, the Standard Update techniques were faster on average then the Partial Updating techniques. With two mobile obstacles, the Standard Updating was even faster. With three mobile obstacles, the Standard Updating was even faster again. An interesting point was that the performance of the different Standard Update techniques was quite varied, by over 260% in a given environment. By contrast, the Partial Updating techniques produced remarkably similar performance results, always staying within 4% of each other.

The point at which the Partial Update became slower then the Standard Updating techniques depended on the ratio of vertices in the domain which were to remain unchanged compared to the number of vertices in the domain which needed to be updated. In larger domains, with mostly static vertices, the

Partial Updating technique would remain the more attractive option for longer then was demonstrated here.

A significant and practical application of the techniques developed in this paper was that of multiple AGVs working together in the same domain at the same time. As was explained in Chapter 5, the ability of an AGV to detect and avoid both stationary and mobile obstacles was an important factor in this.

An automated warehouse could be staffed by multiple AGVs all working in a totally autonomous manner, while each ran it's own instance of the navigation software. The decision to have no relationship between the various AGVs within this domain was an important one. It meant that it would be possible to initially staff a warehouse with a certain number of AGVs and to add more AGVs at a later stage when and if the workload required it. The existing AGVs did not need to be re-programmed in any way to cater for the newly added AGV. They simply treated the new AGV as yet another mobile obstacle in the domain, and continued operating as per normal. This ability to increase the capacity of the overall automated warehouse facility by simple installing more AGVs on the fly was quite important. Given the AGV's ability to avoid both stationary and mobile obstacles. other AGVs in the domain could even be powered down and serviced by humans en situ without having to stop the operation of the other AGVs in the domain. This situation is a vast improvement on the current scenarios encountered in Automated Warehouses, where any maintenance work or modification of the entire system required the entire system to be completely halted for the duration.

What impact would all this have on industry? Without the overriding concern for long term reliability, customers could buy many cheaper AGVs and simply swap in a new AGV in the place of a faulty one, without having to shut down the entire warehouse for the duration. Also. because of the ability to add more AGVs to an existing operation, the "All or Nothing" rule no longer applies. Custom built warehouses which usually take a few years to build and are capable of dealing with the company's planned production for the next 10-15 years are usually only the domain of the multi-national companies. Smaller companies could automate suitable warehouses to a small extend at first, using one or two AGVs in conjunction with the current human staff. Later, these warehouses could be moved to complete automation by simply adding more AGVs while the existing AGVs continued to work uninterrupted. This also meant that companies could continue to operate normally while their warehouses were being automated. another expensive cost avoided. Overall, these points would lower the entry price tag for companies who want to use automated warehouses and would therefore lead to more companies using them.

7. References

CACM = Communications of the Association for Computing Machinery

IEEE = Institute of Electrical and Electronics Engineers

IEEEX = IEEE Expert

IJCAI = International Joint Conference on Artificial Intelligence

IJCNN = International Joint Conference on Neural Networks

IJRA = IEEE Journal of Robotics and Automation

ITRA = IEEE Transactions on Robotics and Automation

PISRR= Proceedings International Symposium on Robotics Research

[ALE83] Aleksander, I, "Reinventing Man", p209-265, Kogan Page, 1983. ISBN#:0-85038-741-8.

[BAK84] Baker,B.S., Bhatt,S.N. & Leighton,T., "An Approximation Algorithm for Manhattan Routing", p205-229, Advances in Computing Research, vol 2, JAI Press, 1984. DCU#:890480532 ISBN#:0-89232-461-9 also p477-486, Proceedings 15th Annual ACM Symposium on the Theory of Computing, 1983

[BEL87] Belter,S.E., "Computer-Aided Routing of Printed Circuit Boards", p200-208, Byte, McGraw-Hill, June 1987.

[BIE92] Biewald,R., "A Neural Controller for Navigation of Non-Holonomic Mobile Robots Using Sensory Information", M.Sc. Thesis, Control Systems Centre, UMIST, Manchester, England, 1992.

[BLU92] Blum, A., "Neural Networks in C++ - An Object Orientated Framework for Building Connectionist Systems", John Wiley & Sons, Inc., 1992. ISBN#:0-471-53847-7.

[BRA84] Brady,M.L. & Brown,D.J., "VLSI Routing: 4 layers suffice", p245-257, Advances in Computing Research, vol 2, JAI Press, 1984. DCU#: 890480532.

[BRO82] Brooks,R., "Solving the findpath problem by good representation of free space", 2nd AAAI Conference, Carnegie Mellon University, August 1982.

[BRO83] Brooks,R. & Lozano-Perez,T., "A subdivision algorithm in Configuration Space for findpath with rotation", Proccedings 8th IJCAI, W.Germany, 1983.

[BRO87] Brooks, R., "Visual Map making for a mobile robot", p438-443, Readings in Computer Vision, Morgan Kaufmann, 1987.

[BUR86] Burnstein,M., "Channel Routing", p133-167, Layout Design and Verification, Elsevier Science Publications, 1986.

[CHE89] Cheung,E. & Lumelsky,V.J., "Proximity Sensing in Robot Manipulator Motion Planning: System and Implementation Issues", p740-751, IEEE Transactions on Robotics and Automation, vol 5, #6, December 1989.

[DON87] Donald,B., "A Search Algorithm for Motion Planning with 6 degrees of freedom", p295-353, Artificial Intelligence, Elsevier Science Publications, March 1987.

[FEN91] Feng,D. & Krogh,B.H., "Dynamic Steering Control of Conventionally Steered Mobile Robots", p699-721, Journal of Robotic Systems, vol 8, #5, October 1991.

[FUJ89] Fujimura,K. and Samet,H., "A Hierarchical Strategy for Path Planning Among Moving Obstacles", p61-69, ITRA, vol 5, #1, February 1989.

[FUN92] Funabiki,N. & Takefuji,Y., "A Parallel Algorithm for Channel Routing Problems", p464-474, IEEE Transactions on Computer Aided Design, vol 11, #4, April 1992.

[GIR84] Giralt,G., Chalia,R. and Vaisset,M., "An integrated navigation and motion control system for autonomous multi-sensory mobile robots", PISRR, MIT Press, 1984.

[GRI90] Griswald,N.C. and Eem,J., "Control for Mobile Robots in the Presence of Moving Objects", p263-268, ITRA, vol 6, #2, April 1990.

[HAW86] Hawker, J.S. et al, "Multiple Robotic Manipulators", p203-219, Byte, McGraw-Hill, January 1986.

[HEM92] Hemami,A., Mehrabi,M.G., & Cheng,R.M.H., "Synthesis of an Optimal Control Law for Path Tracking in Mobile Robots", p383-387, Automatica, vol 28, #2, 1992..

[HEY80] Heynes,W., Sansen,W. & Beke,H., "A line expansion algorithm for the general routing problem with a guarantied solution", p243-249, Proceedings of the 17th Design Automation Conference, June 1980.

[HIG69] Hightower,D., "A solution to the line routing problem on the continuous plane", p1-24, Proceedings of the Design Automation Workshop, 1969.

[HUN91] Hunt,K.J. & Sbarbaro,D., "Neural Networks for nonlinear internal model control", p431-438, IEEE Proceedings of Control Theory and Applications, vol 138, #5, September 1991.

[HUT85] Hu,T.C. & Kuh,E.S., "Theory and Concepts of Circuit Layout", p3-18, Journal of the IEEE, January 1985.

[JOH88] Johnson-Laird, P.N., "The Computer and the Mind", Fontana Masterguides, 1988. ISBN#:0-00-686089-3.

[JON93] Jones, J.L. and Flynn, A.M., "Mobile Robots - Inspiration to Implementation", A. K. Peters, 1993. ISBN#:1-56881-011-3.

[JOR86] Jorgensen,C. et al, "Autonomous Robot Navigation", p223-235, Byte, McGraw-Hill, January 1986.

[KHE88] Kheradpir,S. and Thorp, J.S., "Real Time Control of Robot Manipulators in the Presence of Obstacles", IJRA, vol 4, #6, December 1988.

[KLI90] Klimasauskas, C., "Neural Networks and Image Processing", p77-116, Dr. Dobbs Journal, April 1990.

[KNU73] Knuth, D.E., "The Art of Computer Programming", vols I-III, Addison-Wesley, 1973. ISBN#:0-201-03809-9, 0-201-03822-6, 0-201-03803-X.

[KOS92] Kosko,B., "Neural Networks and Fuzzy Systems", Prentice-Hall International, 1992. ISBN#:0-13-612334-1

[KRA85] Kramer, U., "On the Application of Fuzzy Sets to the Analysis of the System Driver - Vehicle - Environment", p101-107, Automatica, vol 21, #1, 1985.

[KYR92] Kyriakopoulos,K.J. & Saridis,G.N., "Distance Estimation and Collision Prediction for On-Line Robotic Motion Planning", p389-394, Automatica, vol 28, #2, 1992..

[LAU84] Laugier,C. & Pertin,J., "Automatic Grasping: A Case Study in Accessibility Analysis", p201-214, Advanced Software in Robotics, Elsevier Science Publications, 1984.

[LAW90] Lawrence, J.J., "Untangling Neural Nets", p38-44, Dr. Dobbs Journal, April 1990.

[LEV90] Levitt,T.S. & Lawton,D.T., "Qualitative Navigation for Mobile Robots", p305-361, Artificial Intelligence, Elsevier Science Publications, August 1990.

[LUM85] Lumelsky, V.J., "Continuous Robot Motion Planning In Unknown Environment", p339-358, Proceedings of the 4th Workshop on Adaptive Systems Control Theory, Yale University, May 1985.

[MCC88A] McClelland,J. & Rumelhart,D., "Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations", MIT Press, 1988.

[MCC88B] McClelland,J. & Rumelhart,D., "Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 2: Psychological and Biological Models", MIT Press, 1988.

[MCC88C] McClelland,J. & Rumelhart,D., "Explorations in Parallel Distributed Processing - A handbook of Models, Programs and Exercises", MIT Press, 1988. ISBN#:0-262-63113-X.

[MCD84] McDermott,D. & Davis,E., "Planning Routes through Uncertain Territory", p107-155, Artificial Intelligence, Elsevier Science Publications, March 1984.

[MIN69] Minsky,M. & Papert,S., "Perceptrons: Computational Geometry", MIT Press, 1969.

[MOR90] Morris,A.S., "Robot Control", p238-256, Computer Control of Real Time Processes, IEEE, 1990.

[MUL91] Muller,B. & Reinhardt,J., "Neural Networks - An Introduction", Springer-Verlag, 1991.

[NGU89] Nguyen,D.H. & Widrow,B., "The truck backup-upper: An example of self-learning in neural networks", p357-363, Proceedings of the IJCNN, vol 2, June 1989.

[NGU91] Nguyen,D.H. & Widrow,B., "Neural networks for self-learning control systems", p1439-1451, International Journal of Control, vol 54, #6, December 1991.

[NIE91] Niehaus,A. & Stengel,R.F., "An Expert System for Automated Highway Driving", p53-61, IEEE Control Systems, vol 11, #3, April 1991.

[OLI91] Olin,K.E. & Tseng,D.Y., "Autonomous Cross Country Navigation", p16-30, IEEEX, August 1991.

[OZA84] Ozaki,H., Mohri,A. & Takata,M., "On the Collision Free Movement of a Manipulator", p189-200, Advanced Software in Robotics, Elsevier Science Publications, 1984.

[PAU81] Paul,R., "Robot Manipulators - Mathematics, Programming and Control", MIT Press, 1981.

[POM89] Pomerleau,D., "ALVINN: An autonomous land vehicle in a neural network", Advances in Neural Information Processing Systems, Morgan Kaufmann, 1989. ISBN:1-55860-015-9.

[PLU91] Plumer,E.S., "Cascading a Systolic Array and a Feedforward Neural Network for Navigation and Obstacle Avoidance Using Potential Fields", NASA Contractor Report #177575, NASA, 1991.

[RUB74] Rubin,F., "The Lee Connection Algorithm", p907-914, IEEE Trans. Comput, vol C-23, 1974.

[SAI91] Saint-Donat,J., Bhat,N., & McAvoy,T.J., "Neural net based model predictive control", p1453-1468, International Journal of Control, vol 54, #6, December 1991.

[SED90] Sedgewick, R., p389-484, "Algorithms in C", Addison-Wesley, 1990.

[SHI86] Shing,M.T. & Hu,T.C., "Computational Complexity of Layout Problems", p267-293, Layout Design and Verification, Elsevier Science Publications, 1986.

[SOU78] Soukup,J., "Fast Maze Router", p100-102, Proceedings of the 15th Design Automation Conference, Las Vegas, Nevada, June 1978.

[SOU81] Soukup,J., "Circuit Layout", p1281-1304, Proceedings of the IEEE, vol 69, #10, Oct 1981.

[SUH88] Suh,S.H. and Shin,K.G., "A Variational Dynamic Programming Approach to Robot - Path Planning With a Distance Safety Criterion", p334-349, IJRA, vol 4, #3, June 1988.

[TAK89] Takahashi,O. and Schilling,R.J., "Motion Planning in a Plane using Generalized Voronai Diagrams", p143-150, ITRA, vol 5, #2, April 1989.

[THO77] Thompson, A.M., "The Navigation System of the JPL robot", p749-757, Proceedings 5th IJCAI, 1977.

[THO87] Thorpe,C., Herbert,M. et al, "Vision and Navigation for the Carnegie-Mellon NAVLAB", Annual Review of Computer Science, vol 2, Annual Reviews Inc, Palo Alto, CA, 1987.

[THO91] Thorpe,C., Herbert,M., Kanade,T. & Shafer,S., "Towards Autonomous Driving: The CMU Navlab - parts 1& 2", p31-52, IEEEX, August 1991.

[TOU89] Touretzky,D. & Pomerleau,D., "What's hidden in the hidden layers?", p227-233, Byte, McGraw-Hill, August 1989.

[ULL88] Ullman,J., "The PI Placement and Routing System", p425-454, Computational Aspects of VLSI Design, 1988.

[VAN84] VanAken,I.L. & VanBrussel,H., "Software for Solving the Inverse Kinematic Problem for Robot Manipulators in Real Time", p159-174, Advanced Software in Robotics, Elsevier Science Publications, 1984.

[WAN91] Wang,L.L. and Tsal,W.H., "Collision Avoidance by a Modified Least Mean Square Error Classification Scheme for Indoor Autonomous Land Vehicle Navigation", p677-698, Journal of Robotic Systems, vol 8, #5, 1991.

[WAS89] Wasserman,P.D., "Neural Computing - Theory and Practice", Van Nostrand Reinhold, 1989. ISBN#: 0-442-20743-3.

[WAX87] Waxman,A.M. et al, "A Visual Navigation System for Autonomous Land Vehicles", Journal of Robotics and Automation, RA-3, #2, April 1987.

[WIL91] Willis,M.J. et al, "Artificial Neural Networks in Process Engineering", IEEE Proceedings of Control Theory and Applications, vol 138, #3, May 1991.

[WIN84] Winston, P.H., p101-114, "Artificial Intelligence - 2nd ed.", Addison-Wesley, 1984.

[YOS82] Yoshimura,T. & Kuh,E.S., "Efficient Algorithms for Channel Routing", p25-35, IEEE Transactions Computer Aided Design of Integrated Circuits and Systems, vol. CAD-1, January 1982

[ZOM92] Zomaya,A.Y. & Morris,A.S., "Parrallel Computation of Robot Inverse Dynamics for High Speed Motions", p226-236, IEE Proceedings-D, vol. 139, #2, March 1992

A. Benchmarks

In all of the following benchmarks, unless otherwise stated, CPU usage times are measured in seconds to a precision of 1 millisecond. The distance units used in measurements are consistently left unspecified, as the scale of the domain and the scale of the unit distances travelled can be determined by the reader, simply by directly substituting any distance units, from angstroms to furlongs.

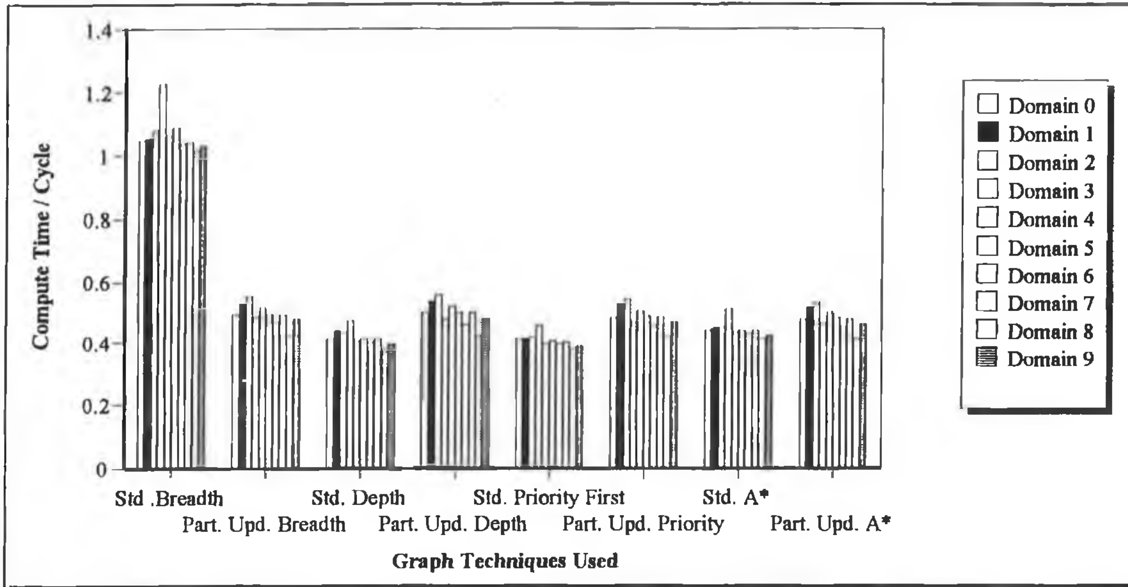
Note that each of the values listed within these tables was the average of ten iterations of each test scenario.

A.1. Single AGV in a Static Domain

			Search Algorithm Used			
			Std .Breadth	Part. Upd. Breadth	Std. Depth	Part. Upd. Depth
0 Mobiles	Map 0	Elapsed Time	47.473	20.278	18.378	20.601
		Compute Time	22.967	10.849	9.111	10.930
		Time Cycles	22	22	22	22
		Distance	23.657	23.657	23.657	23.657
		Compute Time/Cycle	**1.044**	**0.493**	**0.414**	**0.497**
	Map 1	Elapsed Time	31.038	13.434	12.198	13.611
		Compute Time	14.755	7.367	6.144	7.440
		Time Cycles	14	14	14	14
		Distance	15.657	15.657	15.657	15.657
		Compute Time/Cycle	**1.054**	**0.526**	**0.439**	**0.531**
	Map 2	Elapsed Time	62.565	28.373	25.264	28.726
		Compute Time	31.191	15.908	12.560	16.020
		Time Cycles	29	29	29	29
		Distance	31.071	31.071	31.071	31.071
		Compute Time/Cycle	**1.076**	**0.549**	**0.433**	**0.552**
	Map 3	Elapsed Time	12.474	4.537	4.854	4.582
		Compute Time	6.132	2.412	2.362	2.386
		Time Cycles	5	5	5	5
		Distance	5.828	5.828	5.828	5.828
		Compute Time/Cycle	**1.226**	**0.482**	**0.472**	**0.477**
	Map 4	Elapsed Time	44.876	19.750	17.160	20.076
		Compute Time	22.308	10.726	8.576	10.819
		Time Cycles	21	21	21	21
		Distance	23.900	23.900	23.900	23.900
		Compute Time/Cycle	**1.062**	**0.511**	**0.408**	**0.515**
	Map 5	Elapsed Time	28.507	11.986	10.875	12.124
		Compute Time	14.153	6.392	5.385	6.429
		Time Cycles	13	13	13	13
		Distance	15.071	15.071	15.071	15.071
		Compute Time/Cycle	**1.089**	**0.492**	**0.414**	**0.495**
	Map 6	Elapsed Time	26.376	10.721	9.864	10.815
		Compute Time	12.402	5.552	4.805	5.541
		Time Cycles	12	12	12	12
		Distance	12.828	12.828	12.828	12.828
		Compute Time/Cycle	**1.034**	**0.463**	**0.400**	**0.462**
	Map 7	Elapsed Time	43.189	18.467	16.654	18.744
		Compute Time	20.793	9.810	8.213	9.934
		Time Cycles	20	20	20	20
		Distance	22.900	22.900	22.900	22.900
		Compute Time/Cycle	**1.040**	**0.491**	**0.411**	**0.497**
	Map 8	Elapsed Time	21.894	8.513	7.878	8.627
		Compute Time	10.155	4.207	3.800	4.233
		Time Cycles	10	10	10	10
		Distance	11.657	11.657	11.657	11.657
		Compute Time/Cycle	**1.016**	**0.421**	**0.380**	**0.423**
	Map 9	Elapsed Time	38.520	16.280	14.505	16.478
		Compute Time	18.511	8.514	7.137	8.574
		Time Cycles	18	18	18	18
		Distance	20.900	20.900	20.900	20.900
		Compute Time/Cycle	**1.028**	**0.473**	**0.397**	**0.476**

			Search Algorithm Used			
			Std. Priority	Part. Upd. Priority	Std. A*	Part. Upd. A*
0 Mobiles	Map 0	Elapsed Time	18.127	20.681	19.489	20.438
		Compute Time	9.084	10.607	9.626	10.488
		Time Cycles	22	22	22	22
		Distance	23.657	23.657	23.657	23.657
		Compute Time/Cycle	**0.413**	**0.482**	**0.438**	**0.477**
	Map 1	Elapsed Time	11.915	13.698	12.771	13.542
		Compute Time	5.760	7.302	6.247	7.167
		Time Cycles	14	14	14	14
		Distance	15.657	15.657	15.657	15.657
		Compute Time/Cycle	**0.411**	**0.522**	**0.446**	**0.512**
	Map 2	Elapsed Time	24.271	28.869	26.071	28.534
		Compute Time	12.061	15.562	12.966	15.367
		Time Cycles	29	29	29	29
		Distance	31.071	31.071	31.071	31.071
		Compute Time/Cycle	**0.416**	**0.537**	**0.447**	**0.530**
	Map 3	Elapsed Time	4.811	4.620	5.196	4.566
		Compute Time	2.277	2.332	2.524	2.289
		Time Cycles	5	5	5	5
		Distance	5.828	5.828	5.828	5.828
		Compute Time/Cycle	**0.455**	**0.466**	**0.505**	**0.458**
	Map 4	Elapsed Time	16.780	20.113	18.213	19.953
		Compute Time	8.315	10.557	9.064	10.387
		Time Cycles	21	21	21	21
		Distance	23.900	23.900	23.900	23.900
		Compute Time/Cycle	**0.396**	**0.503**	**0.432**	**0.495**
	Map 5	Elapsed Time	10.629	12.204	11.529	12.145
		Compute Time	5.261	6.293	5.711	6.234
		Time Cycles	13	13	13	13
		Distance	15.071	15.071	15.071	15.071
		Compute Time/Cycle	**0.405**	**0.484**	**0.439**	**0.480**
	Map 6	Elapsed Time	9.771	10.935	10.622	10.853
		Compute Time	4.724	5.461	5.162	5.414
		Time Cycles	12	12	12	12
		Distance	12.828	12.828	12.828	12.828
		Compute Time/Cycle	**0.394**	**0.455**	**0.430**	**0.451**
	Map 7	Elapsed Time	16.369	18.811	17.741	18.642
		Compute Time	8.063	9.610	8.720	9.534
		Time Cycles	20	20	20	20
		Distance	22.900	22.900	22.900	22.900
		Compute Time/Cycle	**0.403**	**0.481**	**0.436**	**0.477**
	Map 8	Elapsed Time	7.854	8.751	8.572	8.632
		Compute Time	3.787	4.194	4.097	4.120
		Time Cycles	10	10	10	10
		Distance	11.657	11.657	11.657	11.657
		Compute Time/Cycle	**0.379**	**0.419**	**0.410**	**0.412**
	Map 9	Elapsed Time	14.257	16.653	15.495	16.488
		Compute Time	6.951	8.396	7.597	8.259
		Time Cycles	18	18	18	18
		Distance	20.900	20.900	20.900	20.900
		Compute Time/Cycle	**0.386**	**0.466**	**0.422**	**0.459**

The recorded values for the Compute Time / Cycle for each of the test scenarios, or "Maps", was plotted in the following graph.



The average for each of the recorded Compute Times / Cycle was then calculated as follows:

Averaged Compute Time / Cycle			
Std. Breadth	**Part. Upd. Breadth**	**Std. Depth**	**Part. Upd. Depth**
1.067	0.490	0.417	0.493

Averaged Compute Time / Cycle			
Std. Priority	**Part. Upd. Priority**	**Std. A***	**Part. Upd. A***
0.406	0.482	0.441	0.475

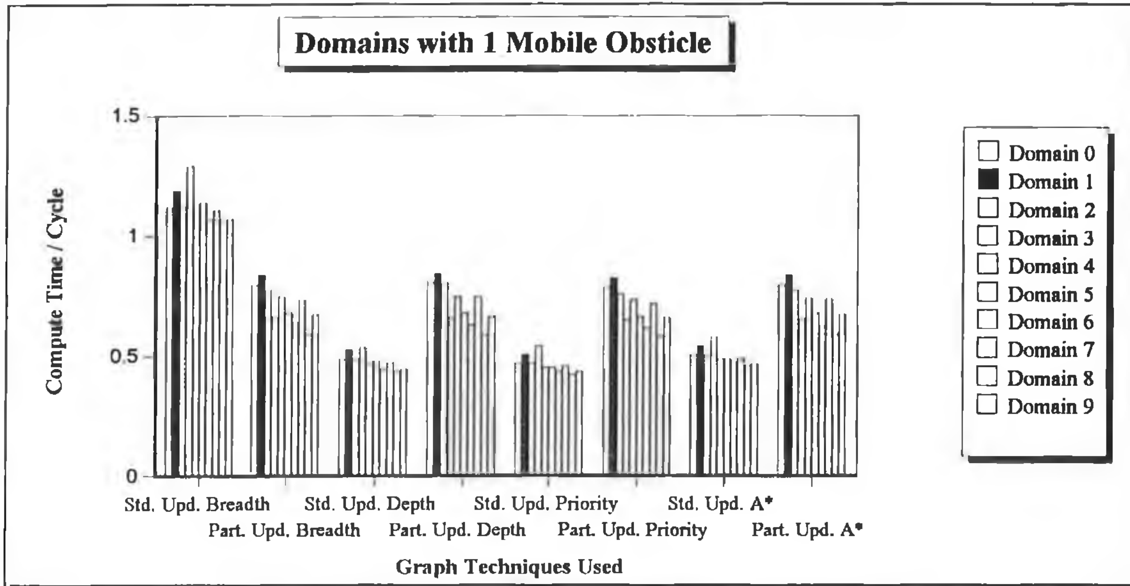
A graph was plotted using these averaged values, and analysed, in Chapter 5.

A.2. Single AGV in a Domain with Independent Mobile Obstacles

			Search Algorithm Used			
			Std. Upd. Breadth	**Part. Upd. Breadth**	**Std. Upd. Depth**	**Part. Upd. Depth**
1 Mobile	Map 0	Elapsed Time	55.118	28.458	23.457	29.001
		Compute Time	26.890	18.303	11.678	18.637
		Time Cycles	24	23	24	23
		Distance	25.047	24.840	25.047	24.840
		Compute Time/Cycle	**1.120**	**0.796**	**0.487**	**0.810**
	Map 1	Elapsed Time	34.509	20.525	14.896	20.905
		Compute Time	16.665	13.385	7.389	13.500
		Time Cycles	14	16	14	16
		Distance	15.940	18.357	15.940	18.357
		Compute Time/Cycle	**1.190**	**0.837**	**0.528**	**0.844**
	Map 2	Elapsed Time	170.105	38.746	73.176	39.965
		Compute Time	84.889	24.675	36.577	25.558
		Time Cycles	76	32	76	32
		Distance	31.130	33.171	31.130	33.171
		Compute Time/Cycle	**1.117**	**0.771**	**0.481**	**0.799**
	Map 3	Elapsed Time	13.199	5.399	5.603	5.450
		Compute Time	6.464	3.287	2.657	3.287
		Time Cycles	5	5	5	5
		Distance	5.828	5.828	5.828	5.828
		Compute Time/Cycle	**1.293**	**0.657**	**0.531**	**0.657**
	Map 4	Elapsed Time	52.480	27.373	21.535	27.798
		Compute Time	26.045	17.096	10.705	17.197
		Time Cycles	23	23	23	23
		Distance	25.124	24.999	25.124	24.999
		Compute Time/Cycle	**1.132**	**0.743**	**0.465**	**0.748**
	Map 5	Elapsed Time	29.838	14.419	12.299	14.509
		Compute Time	14.837	8.844	6.163	8.776
		Time Cycles	13	13	13	13
		Distance	15.071	15.071	15.071	15.071
		Compute Time/Cycle	**1.141**	**0.680**	**0.474**	**0.675**
	Map 6	Elapsed Time	27.282	12.721	10.798	12.819
		Compute Time	12.890	7.615	5.286	7.539
		Time Cycles	12	12	12	12
		Distance	12.828	12.828	12.828	12.828
		Compute Time/Cycle	**1.074**	**0.635**	**0.441**	**0.628**
	Map 7	Elapsed Time	52.096	24.512	21.587	25.012
		Compute Time	25.360	15.424	10.810	15.738
		Time Cycles	23	21	23	21
		Distance	23.241	23.982	23.241	23.982
		Compute Time/Cycle	**1.103**	**0.734**	**0.470**	**0.749**
	Map 8	Elapsed Time	22.793	10.205	8.793	10.284
		Compute Time	10.642	5.903	4.330	5.910
		Time Cycles	10	10	10	10
		Distance	11.657	11.657	11.657	11.657
		Compute Time/Cycle	**1.064**	**0.590**	**0.433**	**0.591**
	Map 9	Elapsed Time	40.110	19.933	16.153	20.115
		Compute Time	19.336	12.078	7.996	11.924
		Time Cycles	18	18	18	18
		Distance	20.900	21.099	20.900	21.099
		Compute Time/Cycle	**1.074**	**0.671**	**0.444**	**0.662**

			Search Algorithm Used			
			Std. Upd. Priorit	**Part. Upd. Priori**	**Std. Upd. A***	**Part. Upd. A***
1 Mobile	Map 0	Elapsed Time	22.778	28.848	24.363	28.990
		Compute Time	11.279	18.056	12.090	18.239
		Time Cycles	24	23	24	23
		Distance	25.047	24.840	25.047	24.840
		Compute Time/Cycle	**0.470**	**0.785**	**0.504**	**0.793**
	Map 1	Elapsed Time	14.312	20.833	15.286	21.009
		Compute Time	7.042	13.214	7.550	13.408
		Time Cycles	14	16	14	16
		Distance	15.940	18.357	15.940	18.357
		Compute Time/Cycle	**0.503**	**0.826**	**0.539**	**0.838**
	Map 2	Elapsed Time	71.330	39.226	76.125	39.575
		Compute Time	35.601	24.273	38.033	24.785
		Time Cycles	76	32	76	32
		Distance	31.130	33.171	31.130	33.171
		Compute Time/Cycle	**0.468**	**0.759**	**0.500**	**0.775**
	Map 3	Elapsed Time	5.527	5.505	5.884	5.510
		Compute Time	2.690	3.235	2.878	3.254
		Time Cycles	5	5	5	5
		Distance	5.828	5.828	5.828	5.828
		Compute Time/Cycle	**0.538**	**0.647**	**0.576**	**0.651**
	Map 4	Elapsed Time	20.887	27.670	22.469	27.797
		Compute Time	10.367	16.811	11.162	16.980
		Time Cycles	23	23	23	23
		Distance	25.124	24.999	25.124	24.999
		Compute Time/Cycle	**0.451**	**0.731**	**0.485**	**0.738**
	Map 5	Elapsed Time	11.961	14.604	12.859	14.691
		Compute Time	5.871	8.664	6.319	8.834
		Time Cycles	13	13	13	13
		Distance	15.071	15.071	15.071	15.071
		Compute Time/Cycle	**0.452**	**0.666**	**0.486**	**0.680**
	Map 6	Elapsed Time	10.709	12.923	11.507	12.963
		Compute Time	5.219	7.407	5.626	7.442
		Time Cycles	12	12	12	12
		Distance	12.828	12.828	12.828	12.828
		Compute Time/Cycle	**0.435**	**0.617**	**0.469**	**0.620**
	Map 7	Elapsed Time	21.073	24.849	22.602	25.035
		Compute Time	10.427	15.156	11.208	15.356
		Time Cycles	23	21	23	21
		Distance	23.241	23.982	23.241	23.982
		Compute Time/Cycle	**0.453**	**0.722**	**0.487**	**0.731**
	Map 8	Elapsed Time	8.737	10.413	9.468	10.453
		Compute Time	4.239	5.828	4.567	5.867
		Time Cycles	10	10	10	10
		Distance	11.657	11.657	11.657	11.657
		Compute Time/Cycle	**0.424**	**0.583**	**0.457**	**0.587**
	Map 9	Elapsed Time	15.818	20.268	17.040	20.400
		Compute Time	7.849	11.857	8.363	12.010
		Time Cycles	18	18	18	18
		Distance	20.900	21.099	20.900	21.099
		Compute Time/Cycle	**0.436**	**0.659**	**0.465**	**0.667**

The recorded values for the Compute Time / Cycle for each of the test scenarios, or "Maps", was plotted in the following graph.

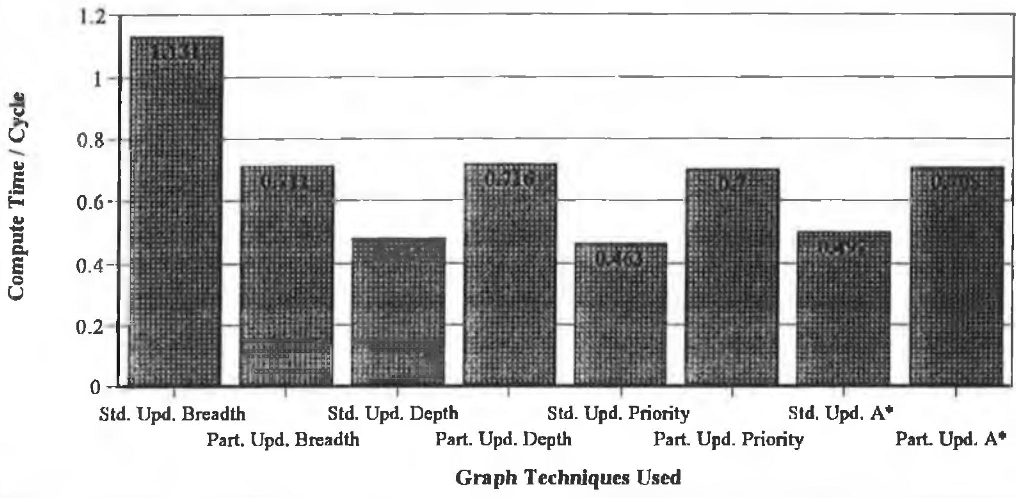


The average for each of the recorded Compute Times / Cycle was then calculated as follows:

Averaged Compute Time / Cycle			
Std. Upd. Breadth	**Part. Upd. Breadth**	**Std. Upd. Depth**	**Part. Upd. Depth**
1.131	0.711	0.475	0.716

Averaged Compute Time / Cycle			
Std. Upd. Priority	**Part. Upd. Priority**	**Std. Upd. A***	**Part. Upd. A***
0.463	0.700	0.497	0.708

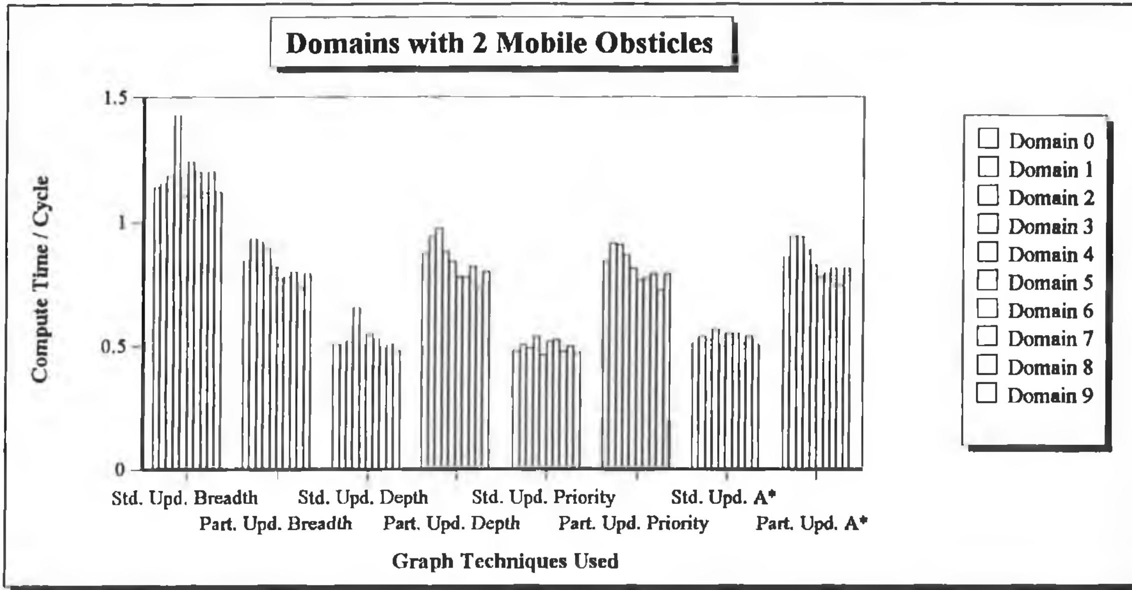
These average values were then plotted to give the following graph.


Domains with 1 Mobile Obsticle (Averaged)
Compute Time / Cycle
1.2
1
0.8
0.6
0.4
0.2
0
0.716
0.463
0.7
Std. Upd. Breadth
Part. Upd. Breadth
Std. Upd. Depth
Part. Upd. Depth
Std. Upd. Priority
Part. Upd. Priority
Std. Upd. A*
Part. Upd. A*
Graph Techniques Used

			Search Algorithm Used			
			Std. Upd. Breadth	**Part. Upd. Breadth**	**Std. Upd. Depth**	**Part. Upd. Depth**
2 Mobiles	Map 0	Elapsed Time	62.396	31.911	27.083	32.994
		Compute Time	30.703	21.039	13.533	21.833
		Time Cycles	27	25	27	25
		Distance	25.412	26.805	25.412	26.905
		Compute Time/Cycle	**1.137**	**0.842**	**0.501**	**0.873**
	Map 1	Elapsed Time	40.141	20.613	34.933	20.968
		Compute Time	19.597	13.937	17.384	14.082
		Time Cycles	17	15	35	15
		Distance	16.057	17.357	16.174	17.357
		Compute Time/Cycle	**1.153**	**0.929**	**0.497**	**0.939**
	Map 2	Elapsed Time	73.720	44.873	30.983	46.888
		Compute Time	36.748	30.318	15.509	32.097
		Time Cycles	31	33	30	33
		Distance	31.354	34.554	31.130	34.554
		Compute Time/Cycle	**1.185**	**0.919**	**0.517**	**0.973**
	Map 3	Elapsed Time	14.638	8.051	6.701	8.130
		Compute Time	7.137	5.328	3.247	5.257
		Time Cycles	5	6	5	6
		Distance	5.970	6.728	5.828	6.728
		Compute Time/Cycle	**1.427**	**0.888**	**0.649**	**0.876**
	Map 4	Elapsed Time	102.396	31.471	22.917	32.278
		Compute Time	50.822	20.396	11.438	20.993
		Time Cycles	46	25	23	25
		Distance	25.665	26.731	25.089	26.731
		Compute Time/Cycle	**1.105**	**0.816**	**0.497**	**0.840**
	Map 5	Elapsed Time	32.614	19.284	14.347	19.584
		Compute Time	16.121	12.308	7.084	12.430
		Time Cycles	13	16	13	16
		Distance	16.144	17.754	16.144	17.754
		Compute Time/Cycle	**1.240**	**0.769**	**0.545**	**0.777**
	Map 6	Elapsed Time	32.099	20.709	13.719	20.818
		Compute Time	15.552	13.245	6.818	13.236
		Time Cycles	13	17	13	17
		Distance	14.891	17.594	14.891	17.594
		Compute Time/Cycle	**1.196**	**0.779**	**0.524**	**0.779**
	Map 7	Elapsed Time	140.180	32.026	60.408	32.621
		Compute Time	69.354	20.773	30.226	21.187
		Time Cycles	62	26	62	26
		Distance	22.958	25.499	22.958	25.499
		Compute Time/Cycle	**1.119**	**0.799**	**0.488**	**0.815**
	Map 8	Elapsed Time	25.347	11.860	10.278	11.967
		Compute Time	11.994	7.381	5.041	7.323
		Time Cycles	10	10	10	10
		Distance	11.915	11.757	11.915	11.757
		Compute Time/Cycle	**1.199**	**0.738**	**0.504**	**0.732**
	Map 9	Elapsed Time	44.023	31.972	18.267	32.241
		Compute Time	21.256	20.616	9.082	20.730
		Time Cycles	19	26	19	26
		Distance	20.900	29.000	20.900	29.000
		Compute Time/Cycle	**1.119**	**0.793**	**0.478**	**0.797**

			Search Algorithm Used			
			Std. Upd. Priorit	Part. Upd. Priori	Std. Upd. A*	Part. Upd. A*
2 Mobiles	Map 0	Elapsed Time	25.820	32.440	27.601	32.923
		Compute Time	12.833	20.899	13.676	21.367
		Time Cycles	27	25	27	25
		Distance	25.412	26.905	25.412	26.905
		Compute Time/Cycle	**0.475**	**0.836**	**0.507**	**0.855**
	Map 1	Elapsed Time	19.268	20.855	20.500	21.125
		Compute Time	9.588	13.682	10.160	14.039
		Time Cycles	19	15	19	15
		Distance	16.233	17.357	16.233	17.357
		Compute Time/Cycle	**0.505**	**0.912**	**0.535**	**0.936**
	Map 2	Elapsed Time	30.540	45.265	32.482	46.241
		Compute Time	15.073	29.833	16.163	31.002
		Time Cycles	31	33	31	33
		Distance	31.071	34.554	31.071	34.554
		Compute Time/Cycle	**0.486**	**0.904**	**0.521**	**0.939**
	Map 3	Elapsed Time	6.590	8.145	7.013	8.258
		Compute Time	3.231	5.170	3.394	5.303
		Time Cycles	6	6	6	6
		Distance	6.111	6.728	6.111	6.728
		Compute Time/Cycle	**0.539**	**0.862**	**0.566**	**0.884**
	Map 4	Elapsed Time	30.825	31.789	32.952	32.260
		Compute Time	15.316	20.181	16.438	20.654
		Time Cycles	33	25	33	25
		Distance	27.021	26.731	27.021	26.731
		Compute Time/Cycle	**0.464**	**0.807**	**0.498**	**0.826**
	Map 5	Elapsed Time	13.522	19.560	14.422	19.779
		Compute Time	6.674	12.184	7.148	12.394
		Time Cycles	13	16	13	16
		Distance	15.802	17.754	15.802	17.754
		Compute Time/Cycle	**0.513**	**0.762**	**0.550**	**0.775**
	Map 6	Elapsed Time	13.745	20.949	14.587	21.301
		Compute Time	6.833	13.075	7.171	13.482
		Time Cycles	13	17	13	17
		Distance	14.774	17.594	14.774	17.594
		Compute Time/Cycle	**0.526**	**0.769**	**0.552**	**0.793**
	Map 7	Elapsed Time	65.989	32.433	70.338	32.850
		Compute Time	32.970	20.580	35.137	21.018
		Time Cycles	69	26	69	26
		Distance	23.899	25.499	23.899	25.499
		Compute Time/Cycle	**0.478**	**0.792**	**0.509**	**0.808**
	Map 8	Elapsed Time	10.249	12.046	10.985	12.117
		Compute Time	4.975	7.242	5.368	7.408
		Time Cycles	10	10	10	10
		Distance	11.915	11.757	11.915	11.757
		Compute Time/Cycle	**0.498**	**0.724**	**0.537**	**0.741**
	Map 9	Elapsed Time	18.132	32.467	19.397	32.922
		Compute Time	8.956	20.483	9.606	21.001
		Time Cycles	19	26	19	26
		Distance	20.900	29.000	20.900	29.000
		Compute Time/Cycle	**0.471**	**0.788**	**0.506**	**0.808**

The recorded values for the Compute Time / Cycle for each of the test scenarios, or "Maps", was plotted in the following graph.

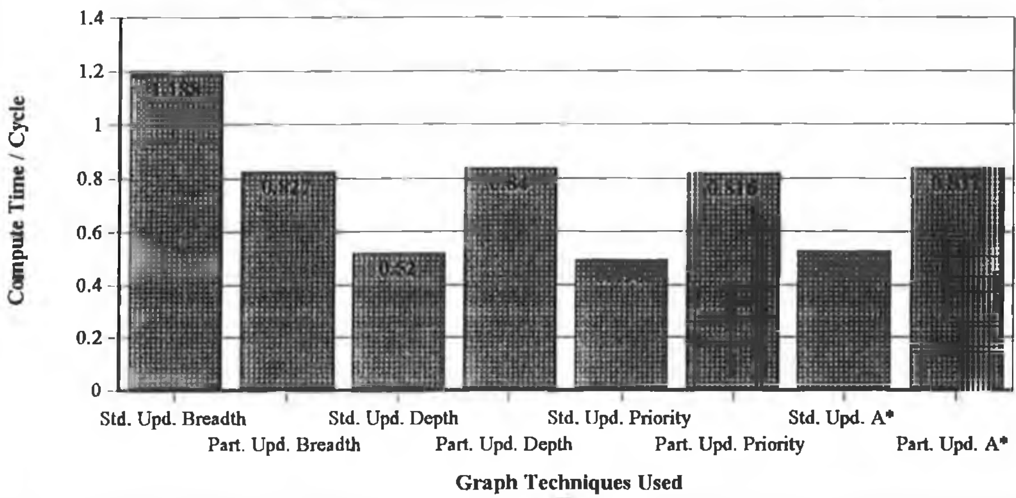


The average for each of the recorded Compute Times / Cycle was then calculated as follows:

Averaged Compute Time / Cycle			
Std. Upd. Breadth	**Part. Upd. Breadth**	**Std. Upd. Depth**	**Part. Upd. Depth**
1.188	**0.827**	**0.52**	**0.84**

Averaged Compute Time / Cycle			
Std. Upd. Priority	**Part. Upd. Priority**	**Std. Upd. A***	**Part. Upd. A***
0.496	**0.816**	**0.528**	**0.837**

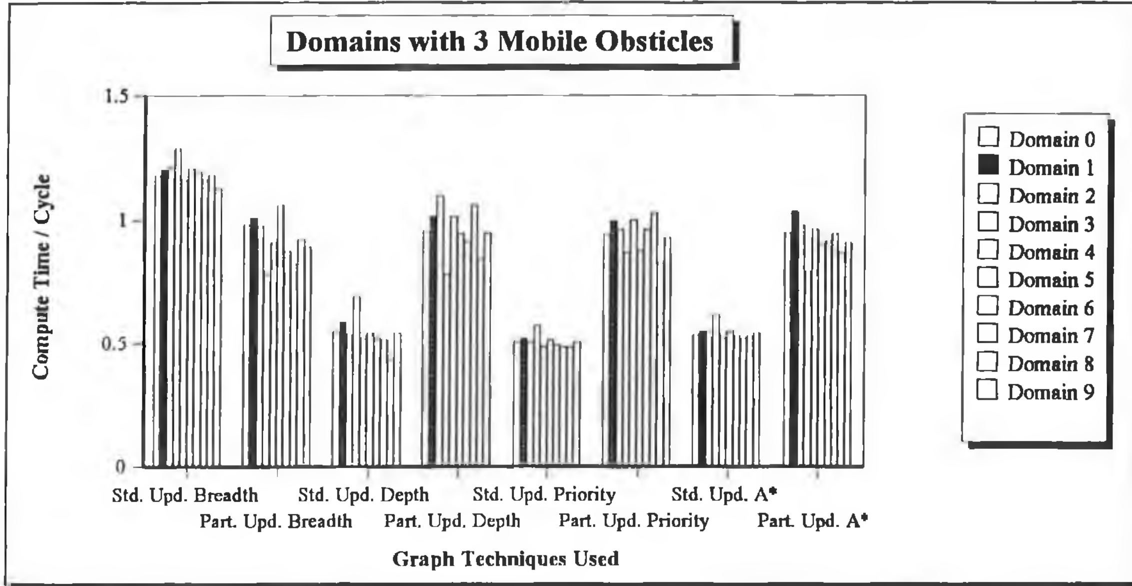
These average values were then plotted to give the following graph.


Domains with 2 Mobile Obsticles (Averaged)
Compute Time / Cycle
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0.827
0.52
0.84
0.816
Std. Upd. Breadth
Part. Upd. Breadth
Std. Upd. Depth
Part. Upd. Depth
Std. Upd. Priority
Part. Upd. Priority
Std. Upd. A*
Part. Upd. A*
Graph Techniques Used

			Search Algorithm Used			
			Std. Upd. Breadth	Part. Upd. Breadth	Std. Upd. Depth	Part. Upd. Depth
3 Mobiles	Map 0	Elapsed Time	64.474	35.488	30.738	43.392
		Compute Time	31.825	24.482	15.264	29.555
		Time Cycles	27	25	28	31
		Distance	26.695	25.771	26.612	27.857
		Compute Time/Cycle	**1.179**	**0.979**	**0.545**	**0.953**
	Map 1	Elapsed Time	46.769	34.758	18.629	26.134
		Compute Time	22.837	24.112	9.268	18.167
		Time Cycles	19	24	16	18
		Distance	16.988	23.057	16.730	18.540
		Compute Time/Cycle	**1.202**	**1.005**	**0.579**	**1.009**
	Map 2	Elapsed Time	75.186	57.668	44.369	52.532
		Compute Time	37.477	39.756	22.101	37.357
		Time Cycles	31	41	42	34
		Distance	31.237	41.678	31.578	36.420
		Compute Time/Cycle	**1.209**	**0.970**	**0.526**	**1.099**
	Map 3	Elapsed Time	18.319	7.239	7.053	7.330
		Compute Time	9.011	4.673	3.402	4.674
		Time Cycles	7	6	5	6
		Distance	6.311	6.028	5.828	6.328
		Compute Time/Cycle	**1.287**	**0.779**	**0.680**	**0.779**
	Map 4	Elapsed Time	75.044	34.982	36.344	39.097
		Compute Time	37.134	23.479	18.065	27.308
		Time Cycles	32	26	35	27
		Distance	28.269	29.214	25.997	28.272
		Compute Time/Cycle	**1.160**	**0.903**	**0.516**	**1.011**
	Map 5	Elapsed Time	39.146	21.148	18.466	22.564
		Compute Time	19.334	14.820	9.087	15.126
		Time Cycles	16	14	17	16
		Distance	16.320	16.454	17.202	17.695
		Compute Time/Cycle	**1.208**	**1.059**	**0.535**	**0.945**
	Map 6	Elapsed Time	32.483	20.812	15.697	23.185
		Compute Time	15.531	13.889	7.779	15.493
		Time Cycles	13	16	15	17
		Distance	13.784	16.577	14.750	17.428
		Compute Time/Cycle	**1.195**	**0.868**	**0.519**	**0.911**
	Map 7	Elapsed Time	63.932	33.883	27.434	37.875
		Compute Time	31.279	22.053	13.705	26.456
		Time Cycles	27	27	27	25
		Distance	24.241	27.082	23.807	26.482
		Compute Time/Cycle	**1.158**	**0.817**	**0.508**	**1.058**
	Map 8	Elapsed Time	27.276	15.176	182.059	14.392
		Compute Time	12.971	10.122	90.947	9.236
		Time Cycles	11	11	212	11
		Distance	12.198	12.223	12.847	11.957
		Compute Time/Cycle	**1.179**	**0.920**	**0.429**	**0.840**
	Map 9	Elapsed Time	51.015	94.027	19.620	70.921
		Compute Time	24.763	63.298	9.676	48.031
		Time Cycles	22	71	18	51
		Distance	23.172	73.899	20.958	54.341
		Compute Time/Cycle	**1.126**	**0.892**	**0.538**	**0.942**

			Search Algorithm Used			
			Std. Upd. Priorit	**Part. Upd. Priori**	**Std. Upd. A***	**Part. Upd. A***
3 Mobiles	Map 0	Elapsed Time	69.360	43.386	73.607	43.648
		Compute Time	34.731	29.117	36.647	29.320
		Time Cycles	69	31	69	31
		Distance	25.661	28.230	25.661	27.857
		Compute Time/Cycle	**0.503**	**0.939**	**0.531**	**0.946**
	Map 1	Elapsed Time	51.878	29.253	54.843	34.175
		Compute Time	25.914	19.813	27.230	23.708
		Time Cycles	50	20	50	23
		Distance	16.140	21.357	16.140	22.598
		Compute Time/Cycle	**0.518**	**0.991**	**0.545**	**1.031**
	Map 2	Elapsed Time	37.771	49.755	40.016	59.826
		Compute Time	18.705	33.496	19.962	40.707
		Time Cycles	37	35	37	42
		Distance	32.061	37.371	32.061	41.878
		Compute Time/Cycle	**0.506**	**0.957**	**0.540**	**0.969**
	Map 3	Elapsed Time	7.104	9.373	7.530	7.453
		Compute Time	3.417	6.025	3.672	4.703
		Time Cycles	6	7	6	6
		Distance	5.970	7.128	5.970	6.028
		Compute Time/Cycle	**0.570**	**0.861**	**0.612**	**0.784**
	Map 4	Elapsed Time	35.059	33.838	37.293	37.054
		Compute Time	17.372	23.008	18.576	24.927
		Time Cycles	36	23	36	26
		Distance	25.089	26.297	25.089	29.389
		Compute Time/Cycle	**0.483**	**1.000**	**0.516**	**0.959**
	Map 5	Elapsed Time	19.618	21.268	20.883	21.850
		Compute Time	9.668	13.958	10.321	14.310
		Time Cycles	19	16	19	16
		Distance	17.251	17.271	17.251	17.178
		Compute Time/Cycle	**0.509**	**0.872**	**0.543**	**0.894**
	Map 6	Elapsed Time	13.024	24.050	13.878	23.372
		Compute Time	6.372	16.305	6.751	15.474
		Time Cycles	13	17	13	17
		Distance	13.925	17.228	13.925	17.628
		Compute Time/Cycle	**0.490**	**0.959**	**0.519**	**0.910**
	Map 7	Elapsed Time	39.628	38.898	42.193	33.829
		Compute Time	19.741	26.657	21.014	22.712
		Time Cycles	41	26	41	24
		Distance	23.748	27.741	23.748	24.999
		Compute Time/Cycle	**0.481**	**1.025**	**0.513**	**0.946**
	Map 8	Elapsed Time	13.722	13.034	14.634	16.088
		Compute Time	6.756	8.257	7.300	10.394
		Time Cycles	14	10	14	12
		Distance	12.774	11.715	12.774	12.357
		Compute Time/Cycle	**0.483**	**0.826**	**0.521**	**0.866**
	Map 9	Elapsed Time	18.249	34.581	19.374	32.663
		Compute Time	9.052	23.144	9.583	21.620
		Time Cycles	18	25	18	24
		Distance	20.900	27.582	20.900	27.199
		Compute Time/Cycle	**0.503**	**0.926**	**0.532**	**0.901**

The recorded values for the Compute Time / Cycle for each of the test scenarios, or "Maps", was plotted in the following graph.

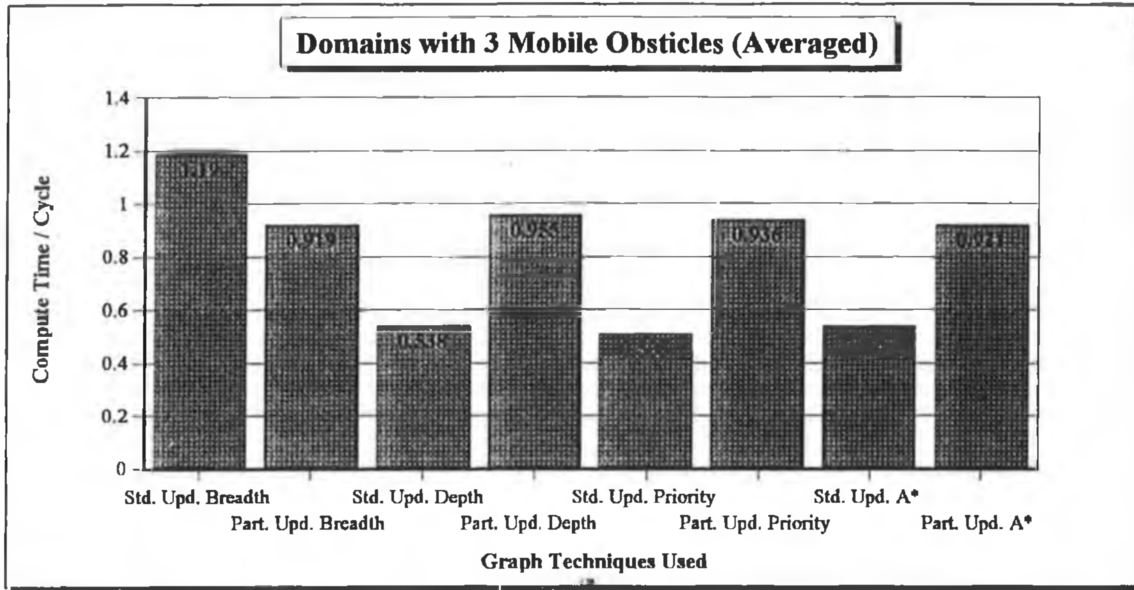


The average for each of the recorded Compute Times / Cycle was then calculated as follows:

Averaged Compute Time / Cycle			
Std. Upd. Breadth	**Part. Upd. Breadth**	**Std. Upd. Depth**	**Part. Upd. Depth**
1.19	0.919	0.538	0.955

Averaged Compute Time / Cycle			
Std. Upd. Priority	**Part. Upd. Priority**	**Std. Upd. A***	**Part. Upd. A***
0.505	0.936	0.537	0.921

These average values were then plotted to give the following graph.



The average values for domains with 1, 2, & 3 mobile obstacles were then averaged as follows.

Averaged Compute Time / Cycle			
Std. Upd. Breadth	**Part. Upd. Breadth**	**Std. Upd. Depth**	**Part. Upd. Depth**
1.17	**0.819**	**0.511**	**0.837**

Averaged Compute Time / Cycle			
Std. Upd. Priority	**Part. Upd. Priority**	**Std. Upd. A***	**Part. Upd. A***
0.488	**0.817**	**0.521**	**0.822**

These averaged values were then plotted and the resulting graph presented in Chapter 5.

B. Source Code Listings

The Standard Lee's Algorithm which was implemented as part of this project was coded in ANSI C. The filenames for each of the separate source code files were supplied inside C format comments (i.e. /* */) at the beginning of each file listing. A detailed explanation of the design behind this program was presented in Chapter 3.

B.1. Standard Lee's Algorithm

```
/* enum.h */
enum directions {front, back, left, right, frontright, frontleft, backright, backleft, up, down, clear};

/* define.h */
#define FALSE 0
#define TRUE !FALSE
/* #define DEBUG */
#define TIMER
#define NUMTIMES 10
#define MAPFILE_MASK "MAP*.DAT"

#define WIDTH 20
#define LENGTH 20
#define HEIGHT 3

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define CLEAR '.'
#define FRONT 'F'
#define BACK 'B'
#define RIGHT 'R'
#define LEFT 'L'
#define BACKLEFT 'T'
#define BACKRIGHT 'U'
#define FRONTLEFT 'V'
#define FRONTRIGHT 'W'
#define DOWN 'D'

#define NOCOST (float)0.0
#define NORMAL (float)1.0
#define NORMAL_DIAG (float)1.414214
#define BLOCKED (float)1000.0

#define PATH '*'

#define ROOM 'R'
#define SCATTER 'S'

#define OBJECT_REC 'O'
#define GOAL_REC 'G'
#define START_REC 'S'

/* struct.h */
typedef struct
	{
	float dist;
	int from;
	char type;
	float cost[9];
	} POINT;

typedef struct daisy
	{
	int w,l,h;
	float dist;
	struct daisy *prev,*next;
	} NODE;

/* proto.h */
void add_node(NODE*);
void calc_benchmark(void);
```

```
int check_back(NODE*);
int check_backleft(NODE*);
int check_backright(NODE*);
int check_front(NODE*);
int check_frontleft(NODE*);
int check_frontright(NODE*);
int check_left(NODE*);
int check_right(NODE*);
int check_up(NODE*);
void del_all_nodes(void);
void del_node(NODE*);
void dump_world(void);
int find_path(void);
int init(char);
int init_once_off(char*);
void init_world(void);
void list_nodes(void);
int main(int, char**);
void mark_around_object(int, int, int);
void mark_path(void);
int position_goal(FILE*);
int position_objects(char, FILE*);
int position_start(FILE*);
void timer_diff(struct timeb*, struct timeb*, struct timeb*);
void timer_init(void);
void timer_log_close(void);
int timer_log_open(char*);
void timer_start(void);
void timer_stop(void);
void trim_list(void);

/* 3dlee.c */
/*************************************************************************

Name: 3dlee.c
Written by: John O'Duinn
Version: 1.0
History: none

This program is my implementation of the standard 2d Lee's algorithm,
extended to deal with time as a 3rd dimension. For the sake of simplicity
only 3 layers (3 time instances) are modelled in this program. However,
the concepts are the same for n time instances.

The basic idea of Lee's algorithm is to find a goal point from a start
point. It does this by checking every point adjacent to the current
point and remembering from what direction the points were reached.
If none of the adjacent points are the goal, it checks their adjacent
points and so on.

Informally, it is very like a pebble in a pond causing ripples which
spread out and eventually reach everywhere in the pond!
*************************************************************************/
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <sys\timeb.h>
#include <dir.h>
#include "struct.h"
#include "define.h"
#include "enum.h"
#include "proto.h"

#ifdef TIMER
struct ffblk cur_mapfile;
FILE* timer_log=(FILE*)NULL;
struct timeb benchmarks[NUMTIMES][2];
#endif
#ifdef DEBUG
FILE* rep;
#endif
POINT world[WIDTH][LENGTH][HEIGHT];
NODE *head;
NODE end;
float trim;
struct timeb orig_time;           /* used to calc elapsed time */
struct timeb timer, start, stop;  /* used to calc compute time */

int main(argc, argv)
int argc;
char **argv;
        {
        int goal_found, i;
        char ch;
```

```
	struct timeb tmp={0L,0,0,0};

	#ifdef TIMER
	if(!init_once_off(argv[0]))
		return(0);
	#endif

	#ifdef DEBUG
	rep = fopen(sprintf("%s.rep", argv[0]),"w");
	if (rep == (FILE*)NULL)
		return(0);
	#endif

	do
		{
		#ifdef TIMER
		for (i=0, tmp.time=0L, tmp.millitm=0; i<NUMTIMES; i++, tmp.time=0L, tmp.millitm=0)
			{
		#endif
			if (argc < 2)
				init(ROOM);
			else
				{
				ch = argv[1][0];
				ch = toupper(ch);
				switch(ch)
					{
					case ROOM:
					case SCATTER:
						init(ch);
						break;
					default:
						init(ROOM);
						break;
					}
				}
			#ifdef TIMER
			fprintf(timer_log, "=====================\n");
			fprintf(timer_log, "Working on %s\n", cur_mapfile.ff_name);
			#endif
			#ifdef DEBUG
			fprintf(rep, "Starting from (%d,%d,%d)\n",
					(*head).w, (*head).l, (*head).h);
			fprintf(rep, "Dest is at (%d,%d,%d)\n",
					end.w, end.l, end.h);
			fflush(rep);
			#endif
			goal_found = find_path();
			if (goal_found)
				{
				printf("\n\nThe distance is %3.1f\n",
						world[end.w][end.l][end.h].dist);
				#ifdef DEBUG
				fprintf(rep, "\n\nThe distance is %3.1f\n",
						world[end.w][end.l][end.h].dist);
				#endif
				}
			else
				{
				printf("Goal not found - Nowhere left to look!\n");
				#ifdef DEBUG
				fprintf(rep, "Goal not found - Nowhere left to look!\n");
				#endif
				}
			#ifdef TIMER
			timer_diff(&orig_time, &stop, &tmp);
			fprintf(timer_log, "%s(%02d) Elapsed time:%05ld.%03d\n", cur_mapfile.ff_name, i,
tmp.time, tmp.millitm);
			benchmarks[i][0].time = tmp.time;
			benchmarks[i][0].millitm = tmp.millitm;
			fprintf(timer_log, "%s(%02d) Compute time:%05ld.%03d\n", cur_mapfile.ff_name, i,
timer.time, timer.millitm);
			benchmarks[i][1].time = timer.time;
			benchmarks[i][1].millitm = timer.millitm;
			fprintf(timer_log, "%s(%02d) Distance to goal:%3.1f\n", cur_mapfile.ff_name, i,
world[end.w][end.l][end.h].dist);
			#endif
			#ifdef DEBUG
			fclose(rep);
			#endif
		#ifdef TIMER
			}
		#endif
		calc_benchmark();
```

```
		} while (!findnext(&cur_mapfile));
	#ifdef TIMER
	if (timer_log != (FILE*)NULL)
		timer_log_close();
	#endif
	return(0);
	}

/* init.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <ctype.h>
#include <dir.h>
#include <sys\timeb.h>
#include "struct.h"
#include "define.h"
#include "enum.h"
#include "proto.h"

#ifdef TIMER
extern FILE* timer_log;
extern struct ffblk cur_mapfile;
extern struct timeb orig_time, timer, start, stop;
#endif
extern float trim;
extern NODE *head, end;
extern POINT world[WIDTH][LENGTH][HEIGHT];


int init(char pattern)
	{
	FILE* fp_data;
	trim = BLOCKED;

	srand((unsigned)time(NULL));
	orig_time.time = timer.time = start.time = stop.time = 0L;
	orig_time.millitm = timer.millitm = start.millitm = stop.millitm = 0;

	fp_data = fopen(cur_mapfile.ff_name, "rt");
	if (fp_data == (FILE*)NULL)
		return(FALSE);

	init_world();
	if (!position_objects(pattern, fp_data))
		{
		fclose(fp_data);
		return(FALSE);
		}
	if (!position_goal(fp_data))
		{
		fclose(fp_data);
		return(FALSE);
		}
	if (!position_start(fp_data))
		{
		fclose(fp_data);
		return(FALSE);
		}

	fclose(fp_data);
	return(TRUE);
	}



int init_once_off(char*argv_zero)
	{
	char mapdir[MAXDIR];

	if (!timer_log_open(argv_zero))
		return(FALSE);

	printf("Enter the directory containing map files, or \"q\" for quit:");
	fflush(stdin);
	scanf("%s", mapdir);
	if (toupper(mapdir[0]) == 'Q')
		{
		printf("Quitting...\n");
		return(FALSE);
		}
	if (chdir(mapdir))
```

```
		{
		printf("The directory %s could not be found.\n", mapdir);
		return(FALSE);
		}
	printf("Made %s the current directory.\n", mapdir);
	if (findfirst(MAPFILE_MASK, &cur_mapfile, 0))
		{
		timer_log_close();
		return(FALSE);
		}
	return(TRUE);
	}


void init_world()
	{
	int a,b,c;

	for(a=0;a<WIDTH;a++)
		{
		for(b=0;b<LENGTH;b++)
			{
			for(c=0;c<HEIGHT;c++)
				{
				world[a][b][c].dist=0.0;
				world[a][b][c].from=clear;
				world[a][b][c].type=CLEAR;
				if (c == HEIGHT-1)
					world[a][b][c].cost[up] = BLOCKED;
				else
					world[a][b][c].cost[up] = NORMAL;
				if ((a == 0) && (b==0))
					{
					world[a][b][c].cost[frontright] = NORMAL_DIAG;
					world[a][b][c].cost[right] = NORMAL;
					world[a][b][c].cost[backright] = BLOCKED;
					world[a][b][c].cost[back] = BLOCKED;
					world[a][b][c].cost[backleft] = BLOCKED;
					world[a][b][c].cost[left] = BLOCKED;
					world[a][b][c].cost[frontleft] = BLOCKED;
					world[a][b][c].cost[front] = NORMAL;
					continue;
					}
				if ((a == 0) && (b < LENGTH-1))
					{
					world[a][b][c].cost[frontright] = NORMAL_DIAG;
					world[a][b][c].cost[right] = NORMAL;
					world[a][b][c].cost[backright] = NORMAL_DIAG;
					world[a][b][c].cost[back] = NORMAL;
					world[a][b][c].cost[backleft] = BLOCKED;
					world[a][b][c].cost[left] = BLOCKED;
					world[a][b][c].cost[frontleft] = BLOCKED;
					world[a][b][c].cost[front] = NORMAL;
					continue;
					}
				if ((a == 0) && (b==LENGTH-1))
					{
					world[a][b][c].cost[frontright] = BLOCKED;
					world[a][b][c].cost[right] = NORMAL;
					world[a][b][c].cost[backright] = NORMAL_DIAG;
					world[a][b][c].cost[back] = NORMAL;
					world[a][b][c].cost[backleft] = BLOCKED;
					world[a][b][c].cost[left] = BLOCKED;
					world[a][b][c].cost[frontleft] = BLOCKED;
					world[a][b][c].cost[front] = BLOCKED;
					continue;
					}
				if ((a < WIDTH-1) && (b == 0))
					{
					world[a][b][c].cost[frontright] = NORMAL_DIAG;
					world[a][b][c].cost[right] = NORMAL;
					world[a][b][c].cost[backright] = BLOCKED;
					world[a][b][c].cost[back] = BLOCKED;
					world[a][b][c].cost[backleft] = BLOCKED;
					world[a][b][c].cost[left] = NORMAL;
					world[a][b][c].cost[frontleft] = NORMAL_DIAG;
					world[a][b][c].cost[front] = NORMAL;
					continue;
					}
				if ((a < LENGTH-1) && (b < WIDTH-1))
					{
					world[a][b][c].cost[frontright] = NORMAL_DIAG;
					world[a][b][c].cost[right] = NORMAL;
```

```
                        world[a][b][c].cost[backright] = NORMAL_DIAG;
                        world[a][b][c].cost[back] = NORMAL;
                        world[a][b][c].cost[backleft] = NORMAL_DIAG;
                        world[a][b][c].cost[left] = NORMAL;
                        world[a][b][c].cost[frontleft] = NORMAL_DIAG;
                        world[a][b][c].cost[front] = NORMAL;
                        continue;
                        }
                if ((a < WIDTH-1) && (b == LENGTH-1))
                        {
                        world[a][b][c].cost[frontright] = BLOCKED;
                        world[a][b][c].cost[right] = NORMAL;
                        world[a][b][c].cost[backright] = NORMAL_DIAG;
                        world[a][b][c].cost[back] = NORMAL;
                        world[a][b][c].cost[backleft] = NORMAL_DIAG;
                        world[a][b][c].cost[left] = NORMAL;
                        world[a][b][c].cost[frontleft] = BLOCKED;
                        world[a][b][c].cost[front] = BLOCKED;
                        continue;
                        }
                if ((a == WIDTH-1) && (b == 0))
                        {
                        world[a][b][c].cost[frontright] = BLOCKED;
                        world[a][b][c].cost[right] = BLOCKED;
                        world[a][b][c].cost[backright] = BLOCKED;
                        world[a][b][c].cost[back] = BLOCKED;
                        world[a][b][c].cost[backleft] = BLOCKED;
                        world[a][b][c].cost[left] = NORMAL;
                        world[a][b][c].cost[frontleft] = NORMAL_DIAG;
                        world[a][b][c].cost[front] = NORMAL;
                        continue;
                        }
                if ((a == WIDTH-1) && (b < WIDTH-1))
                        {
                        world[a][b][c].cost[frontright] = BLOCKED;
                        world[a][b][c].cost[right] = BLOCKED;
                        world[a][b][c].cost[backright] = BLOCKED;
                        world[a][b][c].cost[back] = NORMAL;
                        world[a][b][c].cost[backleft] = NORMAL_DIAG;
                        world[a][b][c].cost[left] = NORMAL;
                        world[a][b][c].cost[frontleft] = NORMAL_DIAG;
                        world[a][b][c].cost[front] = NORMAL;
                        continue;
                        }
                if ((a == LENGTH-1) && (b == WIDTH-1))
                        {
                        world[a][b][c].cost[frontright] = BLOCKED;
                        world[a][b][c].cost[right] = BLOCKED;
                        world[a][b][c].cost[backright] = BLOCKED;
                        world[a][b][c].cost[back] = NORMAL;
                        world[a][b][c].cost[backleft] = NORMAL_DIAG;
                        world[a][b][c].cost[left] = NORMAL;
                        world[a][b][c].cost[frontleft] = BLOCKED;
                        world[a][b][c].cost[front] = BLOCKED;
                        continue;

                }
            }
        }
    }


void mark_around_object(int w, int l, int h)
    {
    int a,b,c;

    for(a=w-1;a<=w+1;a++)
        {
        if ((a < 0) || (a >= WIDTH))
            continue;
        for(b=l-1;b<=l+1;b++)
            {
            if ((b < 0) || (b >= LENGTH))
                continue;
            for(c=h-1;c<=h+1;c++)
                {
                if ((c < 0) || (c >= HEIGHT))
                    continue;

                if ((c != h)
                && ((a != w) || (b != l)))
                    continue;
```

```
                        if ((world[w][l][h].type == OBJECT)
                        && ((world[a][b][c].type == OBJECT)
                        ||  (world[a][b][c].type == ADJ_TO_OBJECT)))
                                continue;

                        if ((world[w][l][h].type == OBJECT)
                        && (world[a][b][c].type == CLEAR))
                                {
                                world[a][b][c].from=clear;
                                world[a][b][c].type=ADJ_TO_OBJECT;
                                }
                        else
                                {
                                world[a][b][c].from=clear;
                                world[a][b][c].type=CLEAR;

                        }
                }
        }
    }


int position_goal(FILE*fp_data)
    {
    char rec_type;
    int i, w, l, h;
    if (fp_data == (FILE*)NULL)
        {
        do
            {
            end.w = rand() % WIDTH;
            end.l = rand() % LENGTH;
            end.h = 0;
            } while (world[end.w][end.l][end.h].type != CLEAR);
        }
    else
        {
        fseek(fp_data, 0L, SEEK_SET);
        while (!feof(fp_data))
            {
            if (fscanf(fp_data, "%c%d%d%d\n", &rec_type, &w, &l, &h) != 4)
                return(FALSE);
            if (rec_type == GOAL_REC)
                {
                end.w = w;
                end.l = l;
                end.h = h;
                break;
                }
            }
        }
    for(i=0;i<HEIGHT;i++)
        {
        world[end.w][end.l][i].from = clear;
        world[end.w][end.l][i].type = GOAL;
        }
    printf("Dest is at (%d,%d,%d)\n",end.w, end.l, end.h);
    return(TRUE);
    }


int position_objects(char pattern, FILE* fp_data)
    {
    char rec_type;
    int i, w, l, h;

    if (pattern == ROOM)
        {
        for(w=0; w<WIDTH-5; w++)
            {
            l=2;
            switch(w)
                {
                case 3:
                case 4:
                case 5:
                    break;
                default:
                    for(h=0; h<HEIGHT; h++)
                        {
                        world[w][l][h].from = clear;
                        world[w][l][h].type = OBJECT;
```

```
                                        mark_around_object(w, l, h);
                                        }
                                break;
                        }
                }
        for(w=0; w<WIDTH-10; w++)
                {
                l=7;
                for(h=0; h<HEIGHT; h++)
                        {
                        world[w][l][h].from = clear;
                        world[w][l][h].type = OBJECT;
                        mark_around_object(w, l, h);
                        }
                }
        for(l=0; l<LENGTH-2; l++)
                {
                w=WIDTH-11;
                switch(l)
                        {
                        case 3:
                        case 4:
                        case 5:
                        case 6:
                        case 7:
                        case 13:
                        case 14:
                        case 15:
                                break;
                        default:
                                for(h=0; h<HEIGHT; h++)
                                        {
                                        world[w][l][h].from = clear;
                                        world[w][l][h].type = OBJECT;
                                        mark_around_object(w, l, h);
                                        }
                                break;
                        }
                }
        for(l=2; l<LENGTH-7; l++)
                {
                w=WIDTH-5;
                switch(l)
                        {
                        case 9:
                        case 10:
                        case 11:
                                break;
                        default:
                                for(h=0; h<HEIGHT; h++)
                                        {
                                        world[w][l][h].from = clear;
                                        world[w][l][h].type = OBJECT;
                                        mark_around_object(w, l, h);
                                        }
                                break;
                        }
                }
        for(w=WIDTH-5; w<WIDTH; w++)
                {
                l=LENGTH-7;
                for(h=0; h<HEIGHT; h++)
                        {
                        world[w][l][h].from = clear;
                        world[w][l][h].type = OBJECT;
                        mark_around_object(w, l, h);
                        }
                }
        for(w=WIDTH-10; w<WIDTH; w++)
                {
                l=LENGTH-3;
                for(h=0; h<HEIGHT; h++)
                        {
                        world[w][l][h].from = clear;
                        world[w][l][h].type = OBJECT;
                        mark_around_object(w, l, h);
                        }
                }
        if (fp_data == (FILE*)NULL)
                {
/*              for(i=0;i< 10; i++)
                        {
                        do
                                }
```

```
                                w = rand() % WIDTH;
                                l = rand() % (LENGTH-HEIGHT-2);
                                h = 0;
                                }
                        while ((world[w][l][h].type != CLEAR)
                                && (world[w][l+1][h].type != CLEAR));
                        for(h=0;h<HEIGHT;h++)
                                {
                                world[w][l+h][h].from = clear;
                                world[w][l+h][h].type = OBJECT;
                                world[w][l+h+1][h].from = clear;
                                world[w][l+h+1][h].type = OBJECT;
                                world[w][l+h+2][h].from = clear;
                                world[w][l+h+2][h].type = OBJECT;
                                }
                        }  */
                }
        else
                {
                fseek(fp_data, 0L, SEEK_SET);
                while (!feof(fp_data))
                        {
                        if ((i = fscanf(fp_data, "%c%d%d%d\n", &rec_type, &w, &l, &h)) != 4)
                                return(FALSE);
                        if (rec_type == OBJECT_REC)
                                {
                                for(h=0;h<HEIGHT;h++)
                                        {
                                        world[w][l+h][h].from = clear;
                                        world[w][l+h][h].type = OBJECT;
                                        mark_around_object(w, l+h, h);
                                        }
                                printf("Object is at (%d,%d,%d)\n",w, l, h);
                                }
                        }
                }
        }
else
        {
        for(l=0;l< 80; l++)
                {
                do
                        {
                        w = rand() % WIDTH;
                        l = rand() % (LENGTH-HEIGHT-1);
                        h = 0;
                        }
                while ((world[w][l][h].type != CLEAR)
                        && (world[w][l+1][h].type != CLEAR));
                for(h=0;h<HEIGHT;h++)
                        {
                        world[w][l+h][h].from = clear;
                        world[w][l+h][h].type = OBJECT;
                        mark_around_object(w, l, h);
                        world[w][l+h+1][h].from = clear;
                        world[w][l+h+1][h].type = OBJECT;
                        mark_around_object(w, l, h);
                        }
                }
        }
return(TRUE);
}


int position_start(FILE*fp_data)
        {
        NODE*new;
        char rec_type;
        int w, l, h;

        if (fp_data == (FILE*)NULL)
                {
                do
                        {
                        w = rand() % WIDTH;
                        l = rand() % LENGTH;
                        h = 0;
                        } while (world[w][l][h].type != CLEAR);
                }
        else
                {
                fseek(fp_data, 0L, SEEK_SET);
                while (!feof(fp_data))
```

```
                {
                if (fscanf(fp_data, "%c%d%d%d\n", &rec_type, &w, &l, &h) != 4)
                        return(FALSE);
                if (rec_type == START_REC)
                        break;
                }
        }
    new = malloc(sizeof(NODE));
    if (new == (NODE*)NULL)
        {
        printf("Unable to start list\n");
        return(FALSE);
        }
    (*new).w = w;
    (*new).l = l;
    (*new).h = h;
    (*new).dist = 0.0;
    (*new).prev=(*new).next=(NODE*)NULL;
    add_node(new);
    world[(*new).w][(*new).l][0].from = clear;
    world[(*new).w][(*new).l][0].type = START;
    printf("Starting from (%d,%d,%d)\n",(*head).w, (*head).l, (*head).h);
    return(TRUE);
    }

/* ll.c */
#include <stdio.h>
#include <stdlib.h>
#include "struct.h"
#include "define.h"
#include "enum.h"
#include "proto.h"

extern FILE* rep;
extern float trim;
extern NODE* head;

void add_node(NODE* new)
    {
    NODE *tmp;

    if (new == (NODE*)NULL)
        return;

    if (head == (NODE*)NULL)
        {
        head = new;
        (*new).prev = (*new).next = new;
        #ifdef DEBUG
        fprintf(rep, "Inserted into empty list (%d,%d,%d)\n",
                        (*new).w, (*new).l, (*new).h);
        #endif
        }
    else
        {
        tmp = head;
        (*new).prev=(*tmp).prev;
        (*tmp).prev=new;

        (*new).next=tmp;
        tmp=(*new).prev;
        (*tmp).next=new;
        #ifdef DEBUG
        fprintf(rep, "Appended (%d,%d,%d)\n",
                        (*new).w, (*new).l, (*new).h);
        #endif
        }
    }


void del_node(NODE*todie)
    {
    NODE *tmp;

    if ((*todie).prev == todie)
        {
        #ifdef DEBUG
        fprintf(rep, "Disposing of Last Node (%d,%d,%d) dist=%3.1f\n",
                (*todie).w, (*todie).l, (*todie).h, (*todie).dist);
        #endif
        head = (NODE*)NULL;
        }
    else
```

```
          {
          tmp=(*todie).prev;
          (*tmp).next = (*todie).next;
          tmp = (*todie).next;
          (*tmp).prev = (*todie).prev;
          head = (*todie).next;
          #ifdef DEBUG
          fprintf(rep, "Disposing of Node (%d,%d,%d) dist=%3.1f\n",
                    (*todie).w, (*todie).l, (*todie).h, (*todie).dist);
          #endif
          }
     free(todie);
     }


void del_all_nodes()
     {
     #ifdef DEBUG
     fprintf(rep, "Deleting entire objects list!\n");
     #endif
     while (head != (NODE*)NULL)
          {
          del_node(head);
          }
     }


void list_nodes()
     {
     #ifdef DEBUG
     NODE *tmp;

     if (head == NULL)
          {
          fprintf(rep, "The objects list is EMPTY!!\n");
          return;
          }
     fprintf(rep, "( w, l, h) Dist Prev     Next    \n");
     fprintf(rep, "---------------------------------\n");
     tmp = head;
     do
          {
          fprintf(rep, "(%2d,%2d,%2d) %3.1f %p %p\n",
                              (*tmp).w, (*tmp).l, (*tmp).h,
                              (*tmp).dist, (*tmp).prev, (*tmp).next);
          tmp = (*tmp).next;
          }
     while (tmp != (NODE*)head);
     fprintf(rep, "-----\n");
     #endif
     }


void trim_list()
     {
     NODE *tmp, *marker;
     int trimmed=TRUE;

     #ifdef DEBUG
     fprintf(rep, "Trimming objects list!\n");
     #endif
     if (head == (NODE*)NULL)
          {
          #ifdef DEBUG
          fprintf(rep, "The objects list is EMPTY!!\n");
          #endif/* DEBUG*/
          return;
          }
     tmp = head;
     marker = (NODE*)NULL;
     while (marker != tmp)
          {
          if (trimmed==TRUE)
               {
               marker=tmp;
               trimmed=FALSE;
               }
          if ((*tmp).dist >= trim)
               {
               #ifdef DEBUG
               fprintf(rep, "(%2d,%2d,%2d) %3.1f %p %p\n",
```

```
                                                        (*tmp).w, (*tmp).l, (*tmp).h,
                                                        (*tmp).dist, (*tmp).prev, (*tmp).next);
                        #endif
                        del_node(tmp);
                        trimmed=TRUE;
                        }
                tmp = (*tmp).next;
                }
        #ifdef DEBUG
        fprintf(rep, "-----\n");
        #endif
        }

/* logfiles.c */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <dir.h>
#include <sys\timeb.h>
#include "struct.h"
#include "define.h"
#include "enum.h"
#include "proto.h"
#ifdef TIMER
extern FILE* timer_log;
#endif
#ifdef DEBUG
extern FILE* rep;
#endif
extern struct timeb orig_time, timer, start, stop;


void timer_log_close()
        {
        fclose(timer_log);
        }


int timer_log_open(char*argv_zero)
        {
        char *tmp;
        char logname[MAXFILE+MAXEXT];

        while((tmp = strchr(argv_zero, '\\')) != (char*)NULL)
                argv_zero = tmp+1;
        strcpy(logname, argv_zero);
        if ((tmp = strchr(logname, '.')) != (char*)NULL)
                strcpy(tmp, ".LOG");
        else
                strcat(logname, ".LOG");
        timer_log = fopen(logname,"wt+");
        if (timer_log == (FILE*)NULL)
                return(FALSE);
        else
                return(TRUE);
        }

/* timer.c */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <sys\timeb.h>
#include "struct.h"
#include "define.h"
#include "enum.h"
#include "proto.h"
#ifdef TIMER
extern FILE* timer_log;
extern struct timeb benchmarks[NUMTIMES][2];
#endif
#ifdef DEBUG
extern FILE* rep;
#endif
extern struct timeb orig_time, timer, start, stop;


void calc_benchmark()
        {
        #ifdef TIMER
        struct timeb avg;
```

```
	int i;

	for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
		{
		avg.time+=benchmarks[i][0].time;
		avg.millitm+=benchmarks[i][0].millitm;
		if (avg.millitm % 1000 != avg.millitm)
			{
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;
			}
		}
	fprintf(timer_log, "Tot. Elapsed time:%05ld.%03d\n", avg.time, avg.millitm);
	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;
	fprintf(timer_log, "Avg. Elapsed time:%05ld.%03d\n", avg.time, avg.millitm);
	for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
		{
		avg.time+=benchmarks[i][1].time;
		avg.millitm+=benchmarks[i][1].millitm;
		if (avg.millitm % 1000 != avg.millitm)
			{
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;
			}
		}
	fprintf(timer_log, "Tot. Compute time:%05ld.%03d\n", avg.time, avg.millitm);
	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;
	fprintf(timer_log, "Avg. Compute time:%05ld.%03d\n", avg.time, avg.millitm);
	fprintf(timer_log, "=====================\n");
	#endif
	}


void timer_diff(struct timeb*start, struct timeb*stop, struct timeb*diff)
	{
	if ((*stop).millitm < (*start).millitm)
		{
		(*stop).millitm += (short)1000; /* carry when subtracting, stops*/
		(*start).time += 1L;            /* negative wraparound problems!*/
		}
	(*diff).millitm += (*stop).millitm - (*start).millitm;
	(*diff).time += (long)((*diff).millitm / (short)1000);
	(*diff).millitm %= (short)1000;
	(*diff).time += ((*stop).time-(*start).time);
	}


void timer_init()
	{
	timer.time = 0L;
	timer.millitm = 0;
	ftime(&orig_time);
	}


void timer_start()
	{
	ftime(&start);
	}


void timer_stop()
	{
	ftime(&stop);
	timer_diff(&start, &stop, &timer);
	}

/* utils.c */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <sys\timeb.h>
#include "struct.h"
#include "define.h"
#include "enum.h"
```

```
#include "proto.h"

extern FILE* rep;
extern float trim;
extern NODE* head;
extern NODE end;
extern POINT world[WIDTH][LENGTH][HEIGHT];
extern struct timeb timer, start, stop;


int check_back(cur)
NODE*cur;
	{
	NODE*new;
	int w, h, l;
	float dist;

	w = (*cur).w;
	l = (*cur).l;
	h = (*cur).h;
	if (world[w][l][h].cost[back] != BLOCKED)
		{
		dist = (*cur).dist;
		dist += world[w][l][h].cost[back];
		l--;
		if ((world[w][l][h].type == OBJECT)
		|| (world[w][l][h].type == ADJ_TO_OBJECT))
			return(FALSE);

		if (dist > trim)
			return(FALSE);
		if (world[w][l][h].type == GOAL)
			{
			world[w][l][h].from = front;
			trim = world[w][l][h].dist - dist;
			return(TRUE);
			}
		if ((world[w][l][h].type == CLEAR)
		&& ((world[w][l][h].from == clear)
			|| (world[w][l][h].dist > dist)))
			{
			new=malloc(sizeof(NODE));
			if (new == (NODE*)NULL)
				{
				printf("Out of Memory!\n");
				exit(0);
				}
			(*new).w = w;
			(*new).l = l;
			(*new).h = h;
			world[w][l][h].dist = (*new).dist = dist;
			world[w][l][h].from = front;
			add_node(new);
			}
		}
	return(FALSE);
	}


int check_backleft(cur)
NODE*cur;
	{
	NODE*new;
	int w, h, l;
	float dist;

	w = (*cur).w;
	l = (*cur).l;
	h = (*cur).h;
	if (world[w][l][h].cost[backleft] != BLOCKED)
		{
		dist = (*cur).dist;
		dist += world[w][l][h].cost[backleft];
		l--;
		w--;
		if ((world[w][l][h].type == OBJECT)
		|| (world[w][l][h].type == ADJ_TO_OBJECT))
			return(FALSE);

		if (dist > trim)
			return(FALSE);
		if (world[w][l][h].type == GOAL)
```

```
			{
			world[w][l][h].from = frontright;
			trim = world[w][l][h].dist = dist;
			return(TRUE);
			}
		if ((world[w][l][h].type == CLEAR)
		&& ((world[w][l][h].from == clear)
			|| (world[w][l][h].dist > dist)))
			{
			new=malloc(sizeof(NODE));
			if (new == (NODE*)NULL)
				{
				printf("Out of Memory!\n");
				exit(0);
				}
			(*new).w = w;
			(*new).l = l;
			(*new).h = h;
			world[w][l][h].dist = (*new).dist = dist;
			world[w][l][h].from = frontright;
			add_node(new);
			}
		}
	return(FALSE);
	}


int check_backright(cur)
NODE*cur;
	{
	NODE*new;
	int w, h, l;
	float dist;

	w = (*cur).w;
	l = (*cur).l;
	h = (*cur).h;
	if (world[w][l][h].cost[backright] != BLOCKED)
		{
		dist = (*cur).dist;
		dist += world[w][l][h].cost[backright];
		l--;
		w++;
		if ((world[w][l][h].type == OBJECT)
		|| (world[w][l][h].type == ADJ_TO_OBJECT))
			return(FALSE);

		if (dist > trim)
			return(FALSE);
		if (world[w][l][h].type == GOAL)
			{
			world[w][l][h].from = frontleft;
			trim = world[w][l][h].dist = dist;
			return(TRUE);
			}
		if ((world[w][l][h].type == CLEAR)
		&& ((world[w][l][h].from == clear)
			|| (world[w][l][h].dist > dist)))
			{
			new=malloc(sizeof(NODE));
			if (new == (NODE*)NULL)
				{
				printf("Out of Memory!\n");
				exit(0);
				}
			(*new).w = w;
			(*new).l = l;
			(*new).h = h;
			world[w][l][h].dist = (*new).dist = dist;
			world[w][l][h].from = frontleft;
			add_node(new);
			}
		}
	return(FALSE);
	}


int check_front(cur)
NODE*cur;
	{
	NODE*new;
	int w, h, l;
```

```
	float dist;

	w = (*cur).w;
	l = (*cur).l;
	h = (*cur).h;
	if (world[w][l][h].cost[front] != BLOCKED)
		{
		dist = (*cur).dist;
		dist += world[w][l][h].cost[front];
		l++;
		if ((world[w][l][h].type == OBJECT)
		|| (world[w][l][h].type == ADJ_TO_OBJECT))
			return(FALSE);

		if (dist > trim)
			return(FALSE);
		if (world[w][l][h].type == GOAL)
			{
			world[w][l][h].from = back;
			trim = world[w][l][h].dist = dist;
			return(TRUE);
			}
		if ((world[w][l][h].type == CLEAR)
		&& ((world[w][l][h].from == clear)
			|| (world[w][l][h].dist > dist)))
			{
			new=malloc(sizeof(NODE));
			if (new == (NODE*)NULL)
				{
				printf("Out of Memory!\n");
				exit(0);
				}
			(*new).w = w;
			(*new).l = l;
			(*new).h = h;
			world[w][l][h].dist = (*new).dist = dist;
			world[w][l][h].from = back;
			add_node(new);
			}
		}
	return(FALSE);
	}


int check_frontleft(cur)
NODE*cur;
	{
	NODE*new;
	int w, h, l;
	float dist;

	w = (*cur).w;
	l = (*cur).l;
	h = (*cur).h;
	if (world[w][l][h].cost[frontleft] != BLOCKED)
		{
		dist = (*cur).dist;
		dist += world[w][l][h].cost[frontleft];
		l++;
		w--;
		if ((world[w][l][h].type == OBJECT)
		|| (world[w][l][h].type == ADJ_TO_OBJECT))
			return(FALSE);

		if (dist > trim)
			return(FALSE);
		if (world[w][l][h].type == GOAL)
			{
			world[w][l][h].from = backright;
			trim = world[w][l][h].dist = dist;
			return(TRUE);
			}
		if ((world[w][l][h].type == CLEAR)
		&& ((world[w][l][h].from == clear)
			|| (world[w][l][h].dist > dist)))
			{
			new=malloc(sizeof(NODE));
			if (new == (NODE*)NULL)
				{
				printf("Out of Memory!\n");
				exit(0);
				}
			(*new).w = w;
```

```
                        (*new).l = l;
                        (*new).h = h;
                        world[w][l][h].dist = (*new).dist = dist;
                        world[w][l][h].from = backright;
                        add_node(new);
                        }
                }
        return(FALSE);
        }


int check_frontright(cur)
NODE*cur;
        {
        NODE*new;
        int w, h, l;
        float dist;

        w = (*cur).w;
        l = (*cur).l;
        h = (*cur).h;
        if (world[w][l][h].cost[frontright] != BLOCKED)
                {
                dist = (*cur).dist;
                dist += world[w][l][h].cost[frontright];
                l++;
                w++;
                if ((world[w][l][h].type == OBJECT)
                || (world[w][l][h].type == ADJ_TO_OBJECT))
                        return(FALSE);

                if (dist > trim)
                        return(FALSE);
                if (world[w][l][h].type == GOAL)
                        {
                        world[w][l][h].from = backleft;
                        trim = world[w][l][h].dist = dist;
                        return(TRUE);
                        }
                if ((world[w][l][h].type == CLEAR)
                && ((world[w][l][h].from == clear)
                        || (world[w][l][h].dist > dist)))
                        {
                        new=malloc(sizeof(NODE));
                        if (new == (NODE*)NULL)
                                {
                                printf("Out of Memory!\n");
                                exit(0);
                                }
                        (*new).w = w;
                        (*new).l = l;
                        (*new).h = h;
                        world[w][l][h].dist = (*new).dist = dist;
                        world[w][l][h].from = backleft;
                        add_node(new);
                        }
                }
        return(FALSE);
        }


int check_left(cur)
NODE*cur;
        {
        NODE*new;
        int w, h, l;
        float dist;

        w = (*cur).w;
        l = (*cur).l;
        h = (*cur).h;
        if (world[w][l][h].cost[left] != BLOCKED)
                {
                dist = (*cur).dist;
                dist += world[w][l][h].cost[left];
                w--;
                if ((world[w][l][h].type == OBJECT)
                || (world[w][l][h].type == ADJ_TO_OBJECT))
                        return(FALSE);

                if (dist > trim)
                        return(FALSE);
```

```
            if (world[w][l][h].type == GOAL)
                  {
                  world[w][l][h].from = right;
                  trim = world[w][l][h].dist = dist;
                  return(TRUE);
                  }
            if ((world[w][l][h].type == CLEAR)
            && ((world[w][l][h].from == clear)
                  || (world[w][l][h].dist > dist)))
                  {
                  new=malloc(sizeof(NODE));
                  if (new == (NODE*)NULL)
                        {
                        printf("Out of Memory!\n");
                        exit(0);
                        }
                  (*new).w = w;
                  (*new).l = l;
                  (*new).h = h;
                  world[w][l][h].dist = (*new).dist = dist;
                  world[w][l][h].from = right;
                  add_node(new);
                  }
      }
      return(FALSE);
      }


int check_right(cur)
NODE*cur;
      {
      NODE*new;
      int w, h, l;
      float dist;

      w = (*cur).w;
      l = (*cur).l;
      h = (*cur).h;
      if (world[w][l][h].cost[right] != BLOCKED)
            {
            dist = (*cur).dist;
            dist += world[w][l][h].cost[right];
            w++;
            if ((world[w][l][h].type == OBJECT)
            || (world[w][l][h].type == ADJ_TO_OBJECT))
                  return(FALSE);

            if (dist > trim)
                  return(FALSE);
            if (world[w][l][h].type == GOAL)
                  {
                  world[w][l][h].from = left;
                  trim = world[w][l][h].dist = dist;
                  return(TRUE);
                  }
            if ((world[w][l][h].type == CLEAR)
            && ((world[w][l][h].from == clear)
                  || (world[w][l][h].dist > dist)))
                  {
                  new=malloc(sizeof(NODE));
                  if (new == (NODE*)NULL)
                        {
                        printf("Out of Memory!\n");
                        exit(0);
                        }
                  (*new).w = w;
                  (*new).l = l;
                  (*new).h = h;
                  world[w][l][h].dist = (*new).dist = dist;
                  world[w][l][h].from = left;
                  add_node(new);
                  }
            }
      return(FALSE);
      }


int check_up(cur)
NODE*cur;
      {
      NODE*new;
      int w, h, l;
      float dist;
```

```
	w = (*cur).w;
	l = (*cur).l;
	h = (*cur).h;
	if ((world[w][l][h].cost[up] != BLOCKED)
	&& (world[w][l][h].from != down))
		{
		dist = (*cur).dist;
		dist += world[w][l][h].cost[up];
		h++;
		if ((world[w][l][h].type == OBJECT)
		|| (world[w][l][h].type == ADJ_TO_OBJECT))
			return(FALSE);

		if (dist > trim)
			return(FALSE);
		if (world[w][l][h].type == GOAL)
			{
			world[w][l][h].from = down;
			trim = world[w][l][h].dist = dist;
			return(TRUE);
			}
		if ((world[w][l][h].type == CLEAR)
		&& ((world[w][l][h].from == clear)
			|| (world[w][l][h].dist > dist)))
			{
			new=malloc(sizeof(NODE));
			if (new == (NODE*)NULL)
				{
				printf("Out of Memory!\n");
				exit(0);
				}
			(*new).w = w;
			(*new).l = l;
			(*new).h = h;
			world[w][l][h].dist = (*new).dist = dist;
			world[w][l][h].from = down;
			add_node(new);
			}
		}
	return(FALSE);
	}


void dump_world()
	{
	int a,b,c, hor,vert;

	for(c=0, hor=1; c<HEIGHT; c++, hor+=25)
		{
		gotoxy(hor,1);
		printf("time=t+%d\n", c);
		for(b=0, vert=2;b<LENGTH;b++, vert++)
			{
			gotoxy(hor, vert);
			for(a=0;a<WIDTH;a++)
				{
				switch(world[a][b][c].type)
					{
					case GOAL:
					case START:
					case OBJECT:
					case ADJ_TO_OBJECT:
					case PATH:
						printf("%c", world[a][b][c].type);
						break;
					case CLEAR:
						switch(world[a][b][c].from)
							{
							case right:
								printf("R");
								break;
							case left:
								printf("L");
								break;
							case up:
								printf("U");
								break;
							case down:
								printf("D");
								break;
							case front:
								printf("F");
```

```
                                                                break;
                                                        case back:
                                                                printf("B");
                                                                break;
                                                        case backleft:
                                                                printf("T");
                                                                break;
                                                        case backright:
                                                                printf("U");
                                                                break;
                                                        case frontleft:
                                                                printf("V");
                                                                break;
                                                        case frontright:
                                                                printf("W");
                                                                break;
                                                        case clear:
                                                        default:
                                                                printf(".");
                                                                break;
                                                        }
                                                break;
                                        }
                                }
                        printf("\n");
                        }
                printf("\n\n");
                }
        }


int find_path()
        {
        NODE *cur;
        int result=FALSE, goal_found=FALSE;

        clrscr();
        timer_init();
        timer_start();
        while (head != (NODE*)NULL)
                {
                timer_stop();
                dump_world();
                timer_start();
                cur = head;
                result = check_front(cur);
                if (result)
                        {
                        goal_found = TRUE;
                        mark_path();
                        #ifdef DEBUG
                        fprintf(rep, "Trim is %3.1f\n", trim);
                        #endif
                        del_node(cur);
                        trim_list();
                        continue;
                        }
                result = check_frontleft(cur);
                if (result)
                        {
                        goal_found = TRUE;
                        mark_path();
                        #ifdef DEBUG
                        fprintf(rep, "Trim is %3.1f\n", trim);
                        #endif
                        del_node(cur);
                        trim_list();
                        continue;
                        }
                result = check_left(cur);
                if (result)
                        {
                        goal_found = TRUE;
                        mark_path();
                        #ifdef DEBUG
                        fprintf(rep, "Trim is %3.1f\n", trim);
                        #endif
                        del_node(cur);
                        trim_list();
                        continue;
                        }
                result = check_backleft(cur);
                if (result)
                        {
```

```
                goal_found = TRUE;
                mark_path();
                #ifdef DEBUG
                fprintf(rep, "Trim is %3.1f\n", trim);
                #endif
                del_node(cur);
                trim_list();
                continue;
                }
            result = check_back(cur);
            if (result)
                {
                goal_found = TRUE;
                mark_path();
                #ifdef DEBUG
                fprintf(rep, "Trim is %3.1f\n", trim);
                #endif
                del_node(cur);
                trim_list();
                continue;
                }
            result = check_backright(cur);
            if (result)
                {
                goal_found = TRUE;
                mark_path();
                #ifdef DEBUG
                fprintf(rep, "Trim is %3.1f\n", trim);
                #endif
                del_node(cur);
                trim_list();
                continue;
                }
            result = check_right(cur);
            if (result)
                {
                goal_found = TRUE;
                mark_path();
                #ifdef DEBUG
                fprintf(rep, "Trim is %3.1f\n", trim);
                #endif
                del_node(cur);
                trim_list();
                continue;
                }
            result = check_frontright(cur);
            if (result)
                {
                goal_found = TRUE;
                mark_path();
                #ifdef DEBUG
                fprintf(rep, "Trim is %3.1f\n", trim);
                #endif
                del_node(cur);
                trim_list();
                continue;
                }
            result = check_up(cur);
            if (result)
                {
                goal_found = TRUE;
                mark_path();
                #ifdef DEBUG
                fprintf(rep, "Trim is %3.1f\n", trim);
                #endif
                del_node(cur);
                trim_list();
                continue;
                }
            del_node(cur);
            }
        timer_stop();
        dump_world();
        return(goal_found);
        }


void mark_path()
        {
        int i, w, h, l;
        float dist;

        w = end.w;
```

```
	l = end.l;
	for(h=0;(h<HEIGHT) && (world[w][l][h].dist==0.0);h++)
		;
	end.h = h;
	dist = world[w][l][h].dist;
	for(i=(int)dist; i>0; i--)
		{
		switch(world[w][l][h].from)
			{
			case front:
				l++;
				break;
			case right:
				w++;
				break;
			case left:
				w--;
				break;
			case back:
				l--;
				break;
			case frontright:
				l++;
				w++;
				break;
			case frontleft:
				l++;
				w--;
				break;
			case backright:
				l--;
				w++;
				break;
			case backleft:
				l--;
				w--;
				break;
			case down:
				h--;
				break;
			default:
				break;
			}
		if (world[w][l][h].type != START)
			world[w][l][h].type = PATH;
		}
	}
```

B.2. Modified Lee's Algorithm

The Modified Lee's Algorithm which was implemented as part of this project was coded in ANSI C. The filenames for each of the separate source code files were supplied inside C format comments (i.e. /* */) at the beginning of each file listing. A detailed explanation of the design behind this program was presented in Chapter 3.

```
/* enum.h */
enum directions {front, back, left, right, frontright, frontleft, backright, backleft,
up, down, clear};

/* define.h */
#define FALSE 0
#define TRUE !FALSE
/* #define DEBUG */
#define TIMER
#define NUMTIMES 3
#define MAPFILE_MASK "MAP*.DAT"

#define WIDTH 20
#define LENGTH 20
#define HEIGHT 3

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define CLEAR '.'
```

```
#define FRONT 'F'
#define BACK 'B'
#define RIGHT 'R'
#define LEFT 'L'
#define BACKLEFT 'T'
#define BACKRIGHT 'U'
#define FRONTLEFT 'V'
#define FRONTRIGHT 'W'
#define DOWN 'D'

#define NOCOST (float)0.0
#define NORMAL (float)1.0
#define NORMAL_DIAG (float)1.414214
#define BLOCKED (float)1000.0

#define PATH '*'

#define ROOM 'R'
#define SCATTER 'S'

#define OBJECT_REC 'O'
#define GOAL_REC 'G'
#define START_REC 'S'

/* struct.h */
typedef struct
        {
        float dist;
        int from;
        char type;
        float cost[9];
        } POINT;

typedef struct daisy
        {
        int w,l,h;
/*      float dist;*/
        float dist_from_goal;
        struct daisy *prev,*next;
        } NODE;

/* proto.h */
NODE* build_node(int, int, int, int, float);
void calc_benchmark(void);
float calc_dist_from_goal(int, int, int);
int check_back(NODE*);
int check_backleft(NODE*);
int check_backright(NODE*);
int check_front(NODE*);
int check_frontleft(NODE*);
int check_frontright(NODE*);
int check_left(NODE*);
int check_line(NODE*);
int check_right(NODE*);
int check_up(NODE*);
void del_all_nodes(void);
void del_node(NODE*);
void dump_world(void);
int find_path(void);
int init(char);
int init_once_off(char*);
void init_world(void);
void insert_node(NODE*);
void list_nodes(void);
int main(int, char**);
void mark_path(void);
int position_goal(FILE*);
int position_objects(char, FILE*);
int position_start(FILE*);
void timer_diff(struct timeb*, struct timeb*, struct timeb*);
void timer_init(void);
void timer_log_close(void);
int  timer_log_open(char*);
void timer_start(void);
void timer_stop(void);
void trim_list(void);

/* 3dleeine.c */
/*******************************************************************

Name: 3dleeine.c
Written by: John O'Duinn
Version: 1.0
History: none
```

```
This program is a further adaptation of my initial 3dlee program. It
attempts to draw a "straight" line from the starting point to the goal,
by moving in whichever of the four directions would reduce the distance
to the goal by the most. This is repeated until

(a) a point is encountered which was already traversed at some earlier
point. If the point was reached in a shorter distance, then the point
is updated and the line continues. If the distance to the point was
further than the previous encounter, than the line terminates (as if a
collision had occurred).

(b) an object is encountered. If this happens, then the 3dlee algorithm
is employed to ripple out from the "collision" and try to find a way
around the object. Every point on the edge of the ripple is repeatedly
used as a starting point for the straight line algorithm and is subject
to the same operating conditions as the initial straight line.

(c) the goal is reached


*****************************************************************/
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <sys\timeb.h>
#include <dir.h>
#include "struct.h"
#include "define.h"
#include "enum.h"
#include "proto.h"

#ifdef TIMER
struct ffblk cur_mapfile;
FILE* timer_log=(FILE*)NULL;
FILE* timer_avg=(FILE*)NULL;
struct timeb benchmarks[NUMTIMES][2];
#endif

#ifdef DEBUG
FILE* rep;
#endif

POINT world[WIDTH][LENGTH][HEIGHT];
NODE *head;
NODE end;
float trim;
struct timeb orig_time;           /* used to calc elapsed time */
struct timeb timer, start, stop;  /* used to calc compute time */

int main(argc, argv)
int argc;
char **argv;
        {
        int goal_found, i;
        char ch;
        struct timeb tmp={0L,0,0,0};

        #ifdef TIMER
        if(!init_once_off(argv[0]))
                return(0);
        #endif

        #ifdef DEBUG
        rep = fopen(sprintf("%s.rep", argv[0]),"w");
        if (rep == (FILE*)NULL)
                return(0);
        #endif

        do
                {
                fprintf(timer_avg, "Working on %s\n", cur_mapfile.ff_name);
                #ifdef TIMER
                for (i=0, tmp.time=0L, tmp.millitm=0; i<NUMTIMES; i++, tmp.time=0L, tmp.millitm=0)
                        {
                #endif
                        if (argc < 2)
                                init(ROOM);
                        else
                                {
                                ch = argv[1][0];
                                ch = toupper(ch);
                                switch(ch)
                                        {
```

```
						case ROOM:
						case SCATTER:
							init(ch);
							break;
						default:
							init(ROOM);
							break;
						|
					|
				#ifdef TIMER
				fprintf(timer_log, "====================\n");
				fprintf(timer_log, "Working on %s\n", cur_mapfile.ff_name);
				#endif
				#ifdef DEBUG
				fprintf(rep, "Starting from (%d,%d,%d)\n",
						(*head).w, (*head).l, (*head).h);
				fprintf(rep, "Dest is at (%d,%d,%d)\n",
						end.w, end.l, end.h);
				fflush(rep);
				#endif
				goal_found = find_path();
				if (goal_found)
					{
					printf("\n\nThe distance is %3.1f\n",
							world[end.w][end.l][end.h].dist);
					#ifdef DEBUG
					fprintf(rep, "\n\nThe distance is %3.1f\n",
							world[end.w][end.l][end.h].dist);
					#endif
					}
				else
					{
					printf("Goal not found - Nowhere left to look!\n");
					#ifdef DEBUG
					fprintf(rep, "Goal not found - Nowhere left to look!\n");
					#endif
					}
				#ifdef TIMER
				timer_diff(&orig_time, &stop, &tmp);
				fprintf(timer_log,  "%s(%02d)  Elapsed  time:%05ld.%03d\n",  cur_mapfile.ff_name,  i,
tmp.time, tmp.millitm);
				benchmarks[i][0].time = tmp.time;
				benchmarks[i][0].millitm = tmp.millitm;
				fprintf(timer_log,  "%s(%02d)  Compute  time:%05ld.%03d\n",  cur_mapfile.ff_name,  i,
timer.time, timer.millitm);
				benchmarks[i][1].time = timer.time;
				benchmarks[i][1].millitm = timer.millitm;
				fprintf(timer_log,            "%s            Distance:%3.1f\n",cur_mapfile.ff_name,
world[end.w][end.l][end.h].dist);
				#endif
				#ifdef DEBUG
				fclose(rep);
				#endif
			#ifdef TIMER
				|
			#endif
			calc_benchmark();
			| while (!findnext(&cur_mapfile));
		#ifdef TIMER
		if (timer_log != (FILE*)NULL)
			timer_log_close();
		#endif
		return(0);
		|

/* init.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <ctype.h>
#include <dir.h>
#include <sys\timeb.h>
#include "struct.h"
#include "define.h"
#include "enum.h"
#include "proto.h"

#ifdef TIMER
extern FILE* timer_log;
extern struct ffblk cur_mapfile;
extern struct timeb orig_time, timer, start, stop;
#endif
```

```
extern float trim;
extern NODE *head, end;
extern POINT world[WIDTH][LENGTH][HEIGHT];


int init(char pattern)
        {
        FILE* fp_data;
        trim = BLOCKED;

        srand((unsigned)time(NULL));
        orig_time.time = timer.time = start.time = stop.time = 0L;
        orig_time.millitm = timer.millitm = start.millitm = stop.millitm = 0;

        fp_data = fopen(cur_mapfile.ff_name, "rt");
        if (fp_data == (FILE*)NULL)
                return(FALSE);

        init_world();
        if (!position_objects(pattern, fp_data))
                {
                fclose(fp_data);
                return(FALSE);
                }
        if (!position_goal(fp_data))
                {
                fclose(fp_data);
                return(FALSE);
                }
        if (!position_start(fp_data))
                {
                fclose(fp_data);
                return(FALSE);
                }

        fclose(fp_data);
        return(TRUE);
        }


int init_once_off(char*argv_zero)
        {
        char mapdir[MAXDIR];

        if (!timer_log_open(argv_zero))
                return(FALSE);

        printf("Enter the directory containing map files, or \"q\" for quit:");
        fflush(stdin);
        scanf("%s", mapdir);
        if (toupper(mapdir[0]) == 'Q')
                {
                printf("Quitting...\n");
                return(FALSE);
                }
        if (chdir(mapdir))
                {
                printf("The directory %s could not be found.\n", mapdir);
                return(FALSE);
                }
        printf("Made %s the current directory.\n", mapdir);
        if (findfirst(MAPFILE_MASK, &cur_mapfile, 0))
                {
                timer_log_close();
                return(FALSE);
                }
        return(TRUE);
        }


void init_world()
        {
        int a,b,c;

        for(a=0;a<WIDTH;a++)
                {
                for(b=0;b<LENGTH;b++)
                        {
                        for(c=0;c<HEIGHT;c++)
                                {
                                world[a][b][c].dist=0.0;
                                world[a][b][c].from=clear;
```

```
world[a][b][c].type=CLEAR;
if (c == HEIGHT-1)
        world[a][b][c].cost[up] = BLOCKED;
else
        world[a][b][c].cost[up] = NORMAL;
if ((a == 0) && (b==0))
        {
        world[a][b][c].cost[frontright] = NORMAL_DIAG;
        world[a][b][c].cost[right] = NORMAL;
        world[a][b][c].cost[backright] = BLOCKED;
        world[a][b][c].cost[back] = BLOCKED;
        world[a][b][c].cost[backleft] = BLOCKED;
        world[a][b][c].cost[left] = BLOCKED;
        world[a][b][c].cost[frontleft] = BLOCKED;
        world[a][b][c].cost[front] = NORMAL;
        continue;
        }
if ((a == 0) && (b < LENGTH-1))
        {
        world[a][b][c].cost[frontright] = NORMAL_DIAG;
        world[a][b][c].cost[right] = NORMAL;
        world[a][b][c].cost[backright] = NORMAL_DIAG;
        world[a][b][c].cost[back] = NORMAL;
        world[a][b][c].cost[backleft] = BLOCKED;
        world[a][b][c].cost[left] = BLOCKED;
        world[a][b][c].cost[frontleft] = BLOCKED;
        world[a][b][c].cost[front] = NORMAL;
        continue;
        }
if ((a == 0) && (b==LENGTH-1))
        {
        world[a][b][c].cost[frontright] = BLOCKED;
        world[a][b][c].cost[right] = NORMAL;
        world[a][b][c].cost[backright] = NORMAL_DIAG;
        world[a][b][c].cost[back] = NORMAL;
        world[a][b][c].cost[backleft] = BLOCKED;
        world[a][b][c].cost[left] = BLOCKED;
        world[a][b][c].cost[frontleft] = BLOCKED;
        world[a][b][c].cost[front] = BLOCKED;
        continue;
        }
if ((a < WIDTH-1) && (b == 0))
        {
        world[a][b][c].cost[frontright] = NORMAL_DIAG;
        world[a][b][c].cost[right] = NORMAL;
        world[a][b][c].cost[backright] = BLOCKED;
        world[a][b][c].cost[back] = BLOCKED;
        world[a][b][c].cost[backleft] = BLOCKED;
        world[a][b][c].cost[left] = NORMAL;
        world[a][b][c].cost[frontleft] = NORMAL_DIAG;
        world[a][b][c].cost[front] = NORMAL;
        continue;
        }
if ((a < LENGTH-1) && (b < WIDTH-1))
        {
        world[a][b][c].cost[frontright] = NORMAL_DIAG;
        world[a][b][c].cost[right] = NORMAL;
        world[a][b][c].cost[backright] = NORMAL_DIAG;
        world[a][b][c].cost[back] = NORMAL;
        world[a][b][c].cost[backleft] = NORMAL_DIAG;
        world[a][b][c].cost[left] = NORMAL;
        world[a][b][c].cost[frontleft] = NORMAL_DIAG;
        world[a][b][c].cost[front] = NORMAL;
        continue;
        }
if ((a < WIDTH-1) && (b == LENGTH-1))
        {
        world[a][b][c].cost[frontright] = BLOCKED;
        world[a][b][c].cost[right] = NORMAL;
        world[a][b][c].cost[backright] = NORMAL_DIAG;
        world[a][b][c].cost[back] = NORMAL;
        world[a][b][c].cost[backleft] = NORMAL_DIAG;
        world[a][b][c].cost[left] = NORMAL;
        world[a][b][c].cost[frontleft] = BLOCKED;
        world[a][b][c].cost[front] = BLOCKED;
        continue;
        }
if ((a == WIDTH-1) && (b == 0))
        {
        world[a][b][c].cost[frontright] = BLOCKED;
        world[a][b][c].cost[right] = BLOCKED;
        world[a][b][c].cost[backright] = BLOCKED;
        world[a][b][c].cost[back] = BLOCKED;
        world[a][b][c].cost[backleft] = BLOCKED;
```

```
                                        world[a][b][c].cost[left] = NORMAL;
                                        world[a][b][c].cost[frontleft] = NORMAL_DIAG;
                                        world[a][b][c].cost[front] = NORMAL;
                                        continue;
                                        }
                                if ((a == WIDTH-1) && (b < WIDTH-1))
                                        {
                                        world[a][b][c].cost[frontright] = BLOCKED;
                                        world[a][b][c].cost[right] = BLOCKED;
                                        world[a][b][c].cost[backright] = BLOCKED;
                                        world[a][b][c].cost[back] = NORMAL;
                                        world[a][b][c].cost[backleft] = NORMAL_DIAG;
                                        world[a][b][c].cost[left] = NORMAL;
                                        world[a][b][c].cost[frontleft] = NORMAL_DIAG;
                                        world[a][b][c].cost[front] = NORMAL;
                                        continue;
                                        }
                                if ((a == LENGTH-1) && (b == WIDTH-1))
                                        {
                                        world[a][b][c].cost[frontright] = BLOCKED;
                                        world[a][b][c].cost[right] = BLOCKED;
                                        world[a][b][c].cost[backright] = BLOCKED;
                                        world[a][b][c].cost[back] = NORMAL;
                                        world[a][b][c].cost[backleft] = NORMAL_DIAG;
                                        world[a][b][c].cost[left] = NORMAL;
                                        world[a][b][c].cost[frontleft] = BLOCKED;
                                        world[a][b][c].cost[front] = BLOCKED;
                                        continue;

                                }
                        }
                }
        }


void mark_around_object(int w, int l, int h)
        {
        int a,b,c;

        for(a=w-1;a<=w+1;a++)
                {
                if ((a < 0) || (a >= WIDTH))
                        continue;
                for(b=l-1;b<=l+1;b++)
                        {
                        if ((b < 0) || (b >= LENGTH))
                                continue;
                        for(c=h-1;c<=h+1;c++)
                                {
                                if ((c < 0) || (c >= HEIGHT))
                                        continue;

                                if ((c != h)
                                && ((a != w) || (b != l)))
                                        continue;

                                if ((world[w][l][h].type == OBJECT)
                                && ((world[a][b][c].type == OBJECT)
                                ||  (world[a][b][c].type == ADJ_TO_OBJECT)))
                                        continue;

                                if ((world[w][l][h].type == OBJECT)
                                && (world[a][b][c].type == CLEAR))
                                        {
                                        world[a][b][c].from=clear;
                                        world[a][b][c].type=ADJ_TO_OBJECT;
                                        }
                                else
                                        {
                                        world[a][b][c].from=clear;
                                        world[a][b][c].type=CLEAR;
                                        }
                                }
                        }
                }
        }


int position_goal(FILE*fp_data)
        {
        char rec_type;
        int i, w, l, h;
```

```
	if (fp_data == (FILE*)NULL)
		{
		do
			{
			end.w = rand() % WIDTH;
			end.l = rand() % LENGTH;
			end.h = 0;
			} while (world[end.w][end.l][end.h].type != CLEAR);
		}
	else
		{
		fseek(fp_data, 0L, SEEK_SET);
		while (!feof(fp_data))
			{
			if (fscanf(fp_data, "%c%d%d%d\n", &rec_type, &w, &l, &h) != 4)
				return(FALSE);
			if (rec_type == GOAL_REC)
				{
				end.w = w;
				end.l = l;
				end.h = h;
				break;
				}
			}
		}
	for(i=0;i<HEIGHT;i++)
		{
		world[end.w][end.l][i].from = clear;
		world[end.w][end.l][i].type = GOAL;
		}
	printf("Dest is at (%d,%d,%d)\n",end.w, end.l, end.h);
	return(TRUE);
	}


int position_objects(char pattern, FILE* fp_data)
	{
	char rec_type;
	int i, w, l, h;

	if (pattern == ROOM)
		{
		for(w=0; w<WIDTH-5; w++)
			{
			l=2;
			switch(w)
				{
				case 3:
				case 4:
				case 5:
					break;
				default:
					for(h=0; h<HEIGHT; h++)
						{
						world[w][l][h].from = clear;
						world[w][l][h].type = OBJECT;
						mark_around_object(w, l, h);
						}
					break;
				}
			}
		for(w=0; w<WIDTH-10; w++)
			{
			l=7;
			for(h=0; h<HEIGHT; h++)
				{
				world[w][l][h].from = clear;
				world[w][l][h].type = OBJECT;
				mark_around_object(w, l, h);
				}
			}
		for(l=0; l<LENGTH-2; l++)
			{
			w=WIDTH-11;
			switch(l)
				{
				case 3:
				case 4:
				case 5:
				case 6:
				case 7:
				case 13:
```

```
				case 14:
				case 15:
					break;
				default:
					for(h=0; h<HEIGHT; h++)
						{
						world[w][l][h].from = clear;
						world[w][l][h].type = OBJECT;
						mark_around_object(w, l, h);
						}
					break;
				}
			}
		for(l=2; l<LENGTH-7; l++)
			{
			w=WIDTH-5;
			switch(l)
				{
				case 9:
				case 10:
				case 11:
					break;
				default:
					for(h=0; h<HEIGHT; h++)
						{
						world[w][l][h].from = clear;
						world[w][l][h].type = OBJECT;
						mark_around_object(w, l, h);
						}
					break;
				}
			}
		for(w=WIDTH-5; w<WIDTH; w++)
			{
			l=LENGTH-7;
			for(h=0; h<HEIGHT; h++)
				{
				world[w][l][h].from = clear;
				world[w][l][h].type = OBJECT;
				mark_around_object(w, l, h);
				}
			}
		for(w=WIDTH-10; w<WIDTH; w++)
			{
			l=LENGTH-3;
			for(h=0; h<HEIGHT; h++)
				{
				world[w][l][h].from = clear;
				world[w][l][h].type = OBJECT;
				mark_around_object(w, l, h);
				}
			}
		if (fp_data == (FILE*)NULL)
			{
/*			for(i=0;i< 10; i++)
				{
				do
					{
					w = rand() % WIDTH;
					l = rand() % (LENGTH-HEIGHT-2);
					h = 0;
					}
				while ((world[w][l][h].type != CLEAR)
					&& (world[w][l+1][h].type != CLEAR));
				for(h=0;h<HEIGHT;h++)
					{
					world[w][l+h][h].from = clear;
					world[w][l+h][h].type = OBJECT;
					world[w][l+h+1][h].from = clear;
					world[w][l+h+1][h].type = OBJECT;
					world[w][l+h+2][h].from = clear;
					world[w][l+h+2][h].type = OBJECT;
					}
				}  */
			}
		else
			{
			fseek(fp_data, 0L, SEEK_SET);
			while (!feof(fp_data))
				{
				if ((i = fscanf(fp_data, "%c%d%d%d\n", &rec_type, &w, &l, &h)) != 4)
					return(FALSE);
				if (rec_type == OBJECT_REC)
					{
```

```
                                        for(h=0;h<HEIGHT;h++)
                                                {
                                                world[w][l+h][h].from = clear;
                                                world[w][l+h][h].type = OBJECT;
                                                mark_around_object(w, l+h, h);
                                                }
                                        printf("Object is at (%d,%d,%d)\n",w, l, h);

                                }
                        }
                }
        else
                {
                for(i=0;i< 80; i++)
                        {
                        do
                                {
                                w = rand() % WIDTH;
                                l = rand() % (LENGTH-HEIGHT-1);
                                h = 0;
                                }
                        while ((world[w][l][h].type != CLEAR)
                                && (world[w][l+1][h].type != CLEAR));
                        for(h=0;h<HEIGHT;h++)
                                {
                                world[w][l+h][h].from = clear;
                                world[w][l+h][h].type = OBJECT;
                                mark_around_object(w, l, h);
                                world[w][l+h+1][h].from = clear;
                                world[w][l+h+1][h].type = OBJECT;
                                mark_around_object(w, l, h);
                                }
                        }
                }
        return(TRUE);
        }


int position_start(FILE*fp_data)
        {
        NODE*new;
        char rec_type;
        int w, l, h;

        if (fp_data == (FILE*)NULL)
                {
                do
                        {
                        w = rand() % WIDTH;
                        l = rand() % LENGTH;
                        h = 0;
                        } while (world[w][l][h].type != CLEAR);
                }
        else
                {
                fseek(fp_data, 0L, SEEK_SET);
                while (!feof(fp_data))
                        {
                        if (fscanf(fp_data, "%c%d%d%d\n", &rec_type, &w, &l, &h) != 4)
                                return(FALSE);
                        if (rec_type == START_REC)
                                break;
                        }
                }
        new = malloc(sizeof(NODE));
        if (new == (NODE*)NULL)
                {
                printf("Unable to start list\n");
                return(FALSE);
                }
        (*new).w = w;
        (*new).l = l;
        (*new).h = h;
        (*new).dist_from_goal = calc_dist_from_goal((*new).w, (*new).l, (*new).h);
        (*new).prev=(*new).next=(NODE*)NULL;
        insert_node(new);
        world[w][l][h].from = clear;
        world[w][l][h].type = START;
        printf("Starting from (%d,%d,%d)\n",(*head).w, (*head).l, (*head).h);
        return(TRUE);
        }

/* ll.c */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "struct.h"
#include "define.h"
#include "enum.h"
#include "proto.h"

#ifdef DEBUG
extern FILE* rep;
#endif
extern float trim;
extern NODE* head;
extern NODE end;
extern POINT world[WIDTH][LENGTH][HEIGHT];


NODE* build_node(int w, int l, int h, int from, float dist)
	{
	float tmp;
	NODE* new;

	if ((w>=0) && (w<WIDTH)
	&& (l>=0) && (l<LENGTH)
	&& (h>=0) && (h<HEIGHT))
		{
		tmp = calc_dist_from_goal(w, l, h);
		if (dist + tmp < trim)
			{
			if ((world[w][l][h].type == CLEAR)
			&& ((world[w][l][h].from == clear)
				|| (world[w][l][h].dist > dist)))
				{
				new=malloc(sizeof(NODE));
				if (new == (NODE*)NULL)
					{
					printf("Out of Memory!\n");
					exit(0);
					}
				(*new).w = w;
				(*new).l = l;
				(*new).h = h;
				(*new).dist_from_goal = tmp;
				(*new).prev = (*new).next = (NODE*)NULL;
				world[w][l][h].dist = dist;
				world[w][l][h].from = from;
				return(new);
				}
			}
		}
	return((NODE*)NULL);
	}


void del_node(NODE*todie)
	{
	NODE *cur;

	cur=(*todie).prev;
	if (cur == todie)
		{
		#ifdef DEBUG
		fprintf(rep, "Disposing of Last Node (%d,%d,%d)\n",
			(*todie).w, (*todie).l, (*todie).h);
		#endif
		head = (NODE*)NULL;
		}
	else
		{
		(*cur).next=(*todie).next;
		cur = (*todie).next;
		(*cur).prev = (*todie).prev;
		if (head == todie)
			head = (*todie).next;
		#ifdef DEBUG
		fprintf(rep, "Disposing of Node (%d,%d,%d)\n",
			(*todie).w, (*todie).l, (*todie).h);
		#endif
		}
	free(todie);
	}
```

```
void del_all_nodes()
	{
	#ifdef DEBUG
	fprintf(rep, "Deleting entire objects list!\n");
	#endif
	while (head != (NODE*)NULL)
		{
		del_node(head);
		}
	}


void insert_node(NODE* new)
	{
	NODE *cur;

	if (new == (NODE*)NULL)
		return;

	if (head == (NODE*)NULL)
		{
		head = new;
		(*new).prev = (*new).next = new;
		#ifdef DEBUG
		fprintf(rep, "Inserted into empty list (%d,%d,%d)\n",
				(*new).w, (*new).l, (*new).h);
		#endif
		return;
		}

	cur=head;
	do
		{
		if ((*cur).dist_from_goal >= (*new).dist_from_goal)
			{
			/* insert before current node */
			if (head == cur)
				head = new;
			(*new).next = cur;
			(*new).prev = (*cur).prev;
			(*cur).prev = new;
			cur = (*new).prev;
			(*cur).next = new;
			return;
			}
		cur = (*cur).next;
		}
	while (cur != head);

	/* has not been inserted yet, so append to dll. */
	cur = head;
	(*new).next = cur;
	(*new).prev = (*cur).prev;
	(*cur).prev = new;
	cur = (*new).prev;
	(*cur).next = new;
	#ifdef DEBUG
	fprintf(rep, "Inserted (%d,%d,%d)\n",
			(*new).w, (*new).l, (*new).h);
	#endif
	}


void list_nodes()
	{
	#ifdef DEBUG
	NODE *tmp;

	if (head == NULL)
		{
		fprintf(rep, "The objects list is EMPTY!!\n");
		return;
		}
	fprintf(rep, "( w, l, h) Dist Prev     Next     \n");
	fprintf(rep, "---------------------------------\n");
	tmp = head;
	do
		{
		fprintf(rep, "(%2d,%2d,%2d) %04d %p %p\n",
				(*tmp).w, (*tmp).l, (*tmp).h,
```

```
                                world[(*tmp).w][(*tmp).l][(*tmp).h].dist,
                                (*tmp).prev, (*tmp).next);
                tmp = (*tmp).next;
                }
        while (tmp != (NODE*)head);
        fprintf(rep, "-----\n");
        #endif
        }


void trim_list()
        {
        float tmp;
        NODE *cur=head, *marker=(NODE*)0;
        int trimmed=TRUE;

        #ifdef DEBUG
        fprintf(rep, "Trimming objects list!\n");
        #endif
        if (cur == (NODE*)NULL)
                {
                #ifdef DEBUG
                fprintf(rep, "The objects list is EMPTY!!\n");
                #endif
                return;
                }
        while ((marker != cur) || (trimmed == TRUE))
                {
                if (trimmed==TRUE)
                        {
                        marker=cur;
                        trimmed=FALSE;
                        }
                tmp = world[(*cur).w][(*cur).l][(*cur).h].dist;
                if ((tmp + (*cur).dist_from_goal) >= trim)
                        {
                        #ifdef DEBUG
                        fprintf(rep, "(%2d,%2d,%2d) %04d %p %p\n",
                                        (*cur).w, (*cur).l, (*cur).h,
                                        tmp, (*cur).prev, (*cur).next);
                        #endif
                        if (cur == (*cur).next)
                                {
                                del_node(cur);
                                break;
                                }
                        else
                                {
                                marker = (*cur).next;
                                del_node(cur);
                                cur=marker;
                                trimmed=TRUE;
                                }
                        }
                else
                        {
                        cur = (*cur).next;
                        }
                }
        #ifdef DEBUG
        fprintf(rep, "-----\n");
        #endif
        }

/* logfiles.c */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <dir.h>
#include <sys\timeb.h>
#include "struct.h"
#include "define.h"
#include "enum.h"
#include "proto.h"
#ifdef TIMER
extern FILE* timer_log;
extern FILE* timer_avg;
#endif
#ifdef DEBUG
extern FILE* rep;
#endif
extern struct timeb orig_time, timer, start, stop;
```

```
void timer_log_close()
        {
        if (timer_log != (FILE*)NULL)
        {
        fclose(timer_log);
        timer_log = (FILE*)NULL;
        }
    if (timer_avg != (FILE*)NULL)
        {
        fclose(timer_avg);
        timer_avg = (FILE*)NULL;
        }
    }


int timer_log_open(char*argv_zero)
        {
        char *tmp;
        char logname[MAXFILE+MAXEXT];

        while((tmp = strchr(argv_zero, '\\')) != (char*)NULL)
                argv_zero = tmp+1;
        strcpy(logname, argv_zero);
        if ((tmp = strchr(logname, '.')) != (char*)NULL)
                strcpy(tmp, ".LOG");
        else
                strcat(logname, ".LOG");
        timer_log = fopen(logname,"wt+");
        if (timer_log == (FILE*)NULL)
                return(FALSE);


        if ((tmp = strchr(logname, '.')) != (char*)NULL)
                strcpy(tmp, ".AVG");
        else
                strcat(logname, ".AVG");
        timer_avg = fopen(logname,"wt+");
        if (timer_avg == (FILE*)NULL)
                return(FALSE);
   else
                return(TRUE);
        }

/* timer.c */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <sys\timeb.h>
#include <dir.h>
#include "struct.h"
#include "define.h"
#include "enum.h"
#include "proto.h"
#ifdef TIMER
extern FILE* timer_log;
extern FILE* timer_avg;
extern struct timeb benchmarks[NUMTIMES][2];
extern struct ffblk cur_mapfile;
#endif
#ifdef DEBUG
extern FILE* rep;
#endif
extern struct timeb orig_time, timer, start, stop;
extern NODE end;
extern POINT world[WIDTH][LENGTH][HEIGHT];



void calc_benchmark()
        {
        #ifdef TIMER
        struct timeb avg;
        int i;

        for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
                {
                avg.time+=benchmarks[i][0].time;
                avg.millitm+=benchmarks[i][0].millitm;
                if (avg.millitm % 1000 != avg.millitm)
                        {
```

```
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;
			}
		}
	fprintf(timer_log, "%s Tot. Elapsed time:%05ld.%03d\n", cur_mapfile.ff_name, avg.time,
avg.millitm);
	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;
	fprintf(timer_avg, "%s Avg. Elapsed time:%05ld.%03d\n", cur_mapfile.ff_name, avg.time,
avg.millitm);
	fprintf(timer_log, "%s Avg. Elapsed time:%05ld.%03d\n", cur_mapfile.ff_name, avg.time,
avg.millitm);
	for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
		{
		avg.time+=benchmarks[i][1].time;
		avg.millitm+=benchmarks[i][1].millitm;
		if (avg.millitm % 1000 != avg.millitm)
			{
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;
			}
		}
	fprintf(timer_log, "%s Tot. Compute time:%05ld.%03d\n", cur_mapfile.ff_name, avg.time,
avg.millitm);
	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;
	fprintf(timer_avg, "%s Avg. Compute time:%05ld.%03d\n", cur_mapfile.ff_name, avg.time,
avg.millitm);
	fprintf(timer_log, "%s Avg. Compute time:%05ld.%03d\n", cur_mapfile.ff_name, avg.time,
avg.millitm);
	fprintf(timer_log, "%s Distance:%3.1f\n", cur_mapfile.ff_name, world[end.w][end.l][end.h].dist);
	fprintf(timer_avg, "%s Distance:%3.1f\n", cur_mapfile.ff_name, world[end.w][end.l][end.h].dist);
	fprintf(timer_avg, "========================\n");
	fprintf(timer_log, "========================\n");
	#endif
	}


void timer_diff(struct timeb*start, struct timeb*stop, struct timeb*diff)
	{
	if ((*stop).millitm < (*start).millitm)
		{
		(*stop).millitm += (short)1000;	/* carry when subtracting, stops*/
		(*start).time += 1L;		/* negative wraparound problems!*/
		}
	(*diff).millitm += (*stop).millitm - (*start).millitm;
	(*diff).time += (long)((*diff).millitm / (short)1000);
	(*diff).millitm %= (short)1000;
	(*diff).time += ((*stop).time-(*start).time);
	}


void timer_init()
	{
	timer.time = 0L;
	timer.millitm = 0;
	ftime(&orig_time);
	}


void timer_start()
	{
	ftime(&start);
	}


void timer_stop()
	{
	ftime(&stop);
	timer_diff(&start, &stop, &timer);
	}

/* utils.c */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <sys\timeb.h>
```

```
#include "struct.h"
#include "define.h"
#include "enum.h"
#include "proto.h"

#ifdef DEBUG
extern FILE* rep;
#endif
extern float trim;
extern NODE* head;
extern NODE end;
extern POINT world[WIDTH][LENGTH][HEIGHT];
extern struct timeb timer, start, stop;


float calc_dist_from_goal(w, l, h)
int w, l;
	{
	float dist=0.0;

	for(;;)
		{
		if (h >0)
			{
			h--;
			dist += NORMAL;
			continue;
			}
		if ((w > end.w) && (l > end.l))
			{
			w--;
			l--;
			dist += NORMAL_DIAG;
			continue;
			}
		if ((w > end.w) && (l == end.l))
			{
			w--;
			dist += NORMAL;
			continue;
			}
		if ((w > end.w) && (l < end.l))
			{
			w--;
			l++;
			dist += NORMAL_DIAG;
			continue;
			}
		if ((w == end.w) && (l > end.l))
			{
			l--;
			dist += NORMAL;
			continue;
			}
		if ((w == end.w) && (l == end.l) && (h == 0))
			break;
		if ((w == end.w) && (l < end.l))
			{
			l++;
			dist += NORMAL;
			continue;
			}
		if ((w < end.w) && (l > end.l))
			{
			w++;
			l--;
			dist += NORMAL_DIAG;
			continue;
			}
		if ((w < end.w) && (l == end.l))
			{
			w++;
			dist += NORMAL;
			continue;
			}
		if ((w < end.w) && (l < end.l))
			{
			w++;
			l++;
			dist += NORMAL_DIAG;
			continue;
			}
		}
```

```
        return(dist);
        }


int check_back(cur)
NODE*cur;
        {
        NODE*new;
        int w, h, l;
        float dist;

        w = (*cur).w;
        l = (*cur).l;
        h = (*cur).h;
        dist = world[w][l][h].dist;

        if ((world[w][l][h].cost[back] == BLOCKED)
        || (world[w][l-1][h].type == ADJ_TO_OBJECT)
        || (world[w][l-1][h].type == OBJECT))
                return(FALSE);

        dist += world[w][l][h].cost[back];
        if (world[w][l-1][h].type == GOAL)
                {
                if ((world[w][l-1][h].from == clear)
                || (dist < world[w][l-1][h].dist))
                        {
                        trim = dist;
                        world[w][l-1][h].dist = trim;
                        world[w][l-1][h].from = front;
                        return(TRUE);
                        }
                else
                        return(FALSE);
                }

        l--;
        if ((world[w][l][h].type == CLEAR)
        && ((world[w][l][h].from == clear)
                || (dist < world[w][l][h].dist)))
                {
                new=build_node(w, l, h, front, dist);
                if (new != (NODE*)NULL)
                        insert_node(new);
                }
        return(FALSE);
        }


int check_backleft(cur)
NODE*cur;
        {
        NODE*new;
        int w, h, l;
        float dist;

        w = (*cur).w;
        l = (*cur).l;
        h = (*cur).h;
        dist = world[w][l][h].dist;

        if ((world[w][l][h].cost[backleft] == BLOCKED)
        || (world[w-1][l-1][h].type == ADJ_TO_OBJECT)
        || (world[w-1][l-1][h].type == OBJECT))
                return(FALSE);

        dist += world[w][l][h].cost[backleft];
        if      (world[w-1][l-1][h].type == GOAL)
                {
                if ((world[w-1][l-1][h].from == clear)
                || (dist < world[w-1][l-1][h].dist))
                        {
                        trim = dist;
                        world[w-1][l-1][h].dist = trim;
                        world[w-1][l-1][h].from = frontright;
                        return(TRUE);
                        }
                else
                        return(FALSE);
                }

        l--;
```

```
        w--;
        if ((world[w][l][h].type == CLEAR)
        && ((world[w][l][h].from == clear)
        || (dist < world[w][l][h].dist)))
                {
                new=build_node(w, l, h, frontright, dist);
                if (new != (NODE*)NULL)
                        insert_node(new);
                }
        return(FALSE);
        }


int check_backright(cur)
NODE*cur;
        {
        NODE*new;
        int w, h, l;
        float dist;

        w = (*cur).w;
        l = (*cur).l;
        h = (*cur).h;
        dist = world[w][l][h].dist;

        if ((world[w][l][h].cost[backright] == BLOCKED)
        || (world[w+1][l-1][h].type == ADJ_TO_OBJECT)
        || (world[w+1][l-1][h].type == OBJECT))
                return(FALSE);

        dist += world[w][l][h].cost[backright];
        if (world[w+1][l-1][h].type == GOAL)
                {
                if ((world[w+1][l-1][h].from == clear)
                || (dist < world[w+1][l-1][h].dist))
                        {
                        trim = dist;
                        world[w+1][l-1][h].dist = trim;
                        world[w+1][l-1][h].from = frontleft;
                        return(TRUE);
                        }
                else
                        return(FALSE);
                }

        l--;
        w++;
        if ((world[w][l][h].type == CLEAR)
        && ((world[w][l][h].from == clear)
        || (dist < world[w][l][h].dist)))
                {
                new=build_node(w, l, h, frontleft, dist);
                if (new != (NODE*)NULL)
                        insert_node(new);
                }
        return(FALSE);
        }


int check_front(cur)
NODE*cur;
        {
        NODE*new;
        int w, h, l;
        float dist;

        w = (*cur).w;
        l = (*cur).l;
        h = (*cur).h;
        dist = world[w][l][h].dist;

        if ((world[w][l][h].cost[front] == BLOCKED)
        || (world[w][l+1][h].type == ADJ_TO_OBJECT)
        || (world[w][l+1][h].type == OBJECT))
                return(FALSE);

        dist += world[w][l][h].cost[front];

        if (world[w][l+1][h].type == GOAL)
                {
                if ((world[w][l+1][h].from == clear)
                || (dist < world[w][l+1][h].dist))
```

```
                        {
                        trim = dist;
                        world[w][l+1][h].dist = trim;
                        world[w][l+1][h].from = back;
                        return(TRUE);
                        }
                else
                        return(FALSE);
                }

        l++;
        if ((world[w][l][h].type == CLEAR)
        && ((world[w][l][h].from == clear)
        || (dist < world[w][l][h].dist)))
                {
                new=build_node(w, l, h, back, dist);
                if (new != (NODE*)NULL)
                        insert_node(new);
                }
        return(FALSE);
        }


int check_frontleft(cur)
NODE*cur;
        {
        NODE*new;
        int w, h, l;
        float dist;

        w = (*cur).w;
        l = (*cur).l;
        h = (*cur).h;
        dist = world[w][l][h].dist;

        if ((world[w][l][h].cost[frontleft] == BLOCKED)
        || (world[w-1][l+1][h].type == ADJ_TO_OBJECT)
        || (world[w-1][l+1][h].type == OBJECT))
                return(FALSE);

        dist += world[w][l][h].cost[frontleft];

        if (world[w-1][l+1][h].type == GOAL)
                {
                if ((world[w-1][l+1][h].from == clear)
                || (dist < world[w-1][l+1][h].dist))
                        {
                        trim = dist;
                        world[w-1][l+1][h].dist = trim;
                        world[w-1][l+1][h].from = backright;
                        return(TRUE);
                        }
                else
                        return(FALSE);
                }

        l++;
        w--;
        if ((world[w][l][h].type == CLEAR)
        && ((world[w][l][h].from == clear)
        || (dist < world[w][l][h].dist)))
                {
                new=build_node(w, l, h, backright, dist);
                if (new != (NODE*)NULL)
                        insert_node(new);
                }
        return(FALSE);
        }


int check_frontright(cur)
NODE*cur;
        {
        NODE*new;
        int w, h, l;
        float dist;

        w = (*cur).w;
        l = (*cur).l;
        h = (*cur).h;
        dist = world[w][l][h].dist;
```

```
        if ((world[w][l][h].cost[frontright] == BLOCKED)
        || (world[w+1][l+1][h].type == ADJ_TO_OBJECT)
        || (world[w+1][l+1][h].type == OBJECT))
                return(FALSE);

        dist += world[w][l][h].cost[frontright];

        if (world[w+1][l+1][h].type == GOAL)
                {
                if ((world[w+1][l+1][h].from == clear)
                || (dist < world[w+1][l+1][h].dist))
                        {
                        trim = dist;
                        world[w+1][l+1][h].dist = trim;
                        world[w+1][l+1][h].from = back;
                        return(TRUE);
                        }
                else
                        return(FALSE);
                }

        l++;
        w++;
        if ((world[w][l][h].type == CLEAR)
        && ((world[w][l][h].from == clear)
        || (dist < world[w][l][h].dist)))
                {
                new=build_node(w, l, h, backleft, dist);
                if (new != (NODE*)NULL)
                        insert_node(new);
                }
        return(FALSE);
        }


int check_left(cur)
NODE*cur;
        {
        NODE*new;
        int w, h, l;
        float dist;

        w = (*cur).w;
        l = (*cur).l;
        h = (*cur).h;
        dist = world[w][l][h].dist;

        if ((world[w][l][h].cost[left] == BLOCKED)
        || (world[w-1][l][h].type == ADJ_TO_OBJECT)
        || (world[w-1][l][h].type == OBJECT))
                return(FALSE);

        dist += world[w][l][h].cost[left];

        if (world[w-1][l][h].type == GOAL)
                {
                if ((world[w-1][l][h].from == clear)
                || (dist < world[w-1][l][h].dist))
                        {
                        trim = dist;
                        world[w-1][l][h].dist = trim;
                        world[w-1][l][h].from = right;
                        return(TRUE);
                        }
                else
                        return(FALSE);
                }

        w--;
        if ((world[w][l][h].type == CLEAR)
        && ((world[w][l][h].from == clear)
        || (dist < world[w][l][h].dist)))
                {
                new=build_node(w, l, h, right, dist);
                if (new != (NODE*)NULL)
                        insert_node(new);
                }
        return(FALSE);
        }


int check_line(cur)
NODE*cur;
```

```
{
NODE*new;
int tmp_w, tmp_l, w, h, l, from;
float dist;

w = (*cur).w;
l = (*cur).l;
h = (*cur).h;
dist = world[w][l][h].dist;

for(;;)
        {
        tmp_w = end.w - w;
        tmp_l = end.l - l;
        if ((tmp_w > 0)
        && (tmp_l > 0))
                {
                from = backleft;
                dist += world[w][l][h].cost[frontright];
                w++;
                l++;
                }
        if ((tmp_w > 0)
        && (tmp_l == 0))
                {
                from = left;
                dist += world[w][l][h].cost[right];
                w++;
                }
        if ((tmp_w > 0)
        && (tmp_l < 0))
                {
                from = frontleft;
                dist += world[w][l][h].cost[backright];
                w++;
                l--;
                }
        if ((tmp_w < 0)
        && (tmp_l > 0))
                {
                from = backright;
                dist += world[w][l][h].cost[frontleft];
                w--;
                l++;
                }
        if ((tmp_w < 0)
        && (tmp_l == 0))
                {
                from = right;
                dist += world[w][l][h].cost[left];
                w--;
                }
        if ((tmp_w < 0)
        && (tmp_l < 0))
                {
                from = frontright;
                dist += world[w][l][h].cost[backleft];
                w--;
                l--;
                }
        if ((tmp_w == 0)
        && (tmp_l > 0))
                {
                from = back;
                dist += world[w][l][h].cost[front];
                l++;
                }
        if ((tmp_w == 0)
        && (tmp_l < 0))
                {
                from = front;
                dist += world[w][l][h].cost[back];
                l--;
                }
        if (world[w][l][h].type == GOAL)
                {
                if ((world[w][l][h].from == clear)
                || (dist < world[w][l][h].dist))
                        {
                        trim = dist;
                        world[w][l][h].dist = dist;
                        world[w][l][h].from = from;
                        return(TRUE);
                        }
```

```
			else
				return(FALSE);
			}
		if ((world[w][l][h].type == OBJECT)
		|| (world[w][l][h].type == ADJ_TO_OBJECT))
			return(FALSE);

		if ((world[w][l][h].type == CLEAR)
		&& ((world[w][l][h].from == clear)
		|| (dist < world[w][l][h].dist)))
			{
			new = build_node(w, l, h, from, dist);
			if (new != (NODE*)NULL)
				insert_node(new);
			else
				return(FALSE);
			}
		dump_world();
		}
	}


int check_right(cur)
NODE*cur;
	{
	NODE*new;
	int w, h, l;
	float dist;

	w = (*cur).w;
	l = (*cur).l;
	h = (*cur).h;
	dist = world[w][l][h].dist;

	if ((world[w][l][h].cost[right] == BLOCKED)
	|| (world[w+1][l][h].type == ADJ_TO_OBJECT)
	|| (world[w+1][l][h].type == OBJECT))
		return(FALSE);

	dist += world[w][l][h].cost[right];

	if (world[w+1][l][h].type == GOAL)
		{
		if ((world[w+1][l][h].from == clear)
		|| (dist < world[w+1][l][h].dist))
			{
			trim = dist;
			world[w+1][l][h].dist = trim;
			world[w+1][l][h].from = left;
			return(TRUE);
			}
		else
			return(FALSE);
		}

	w++;
	if ((world[w][l][h].type == CLEAR)
	&& ((world[w][l][h].from == clear)
	|| (dist < world[w][l][h].dist)))
		{
		new=build_node(w, l, h, left, dist);
		if (new != (NODE*)NULL)
			insert_node(new);
		}
	return(FALSE);
	}


int check_up(cur)
NODE*cur;
	{
	NODE*new;
	int w, h, l;
	float dist;

	w = (*cur).w;
	l = (*cur).l;
	h = (*cur).h;
	dist = world[w][l][h].dist;

	if ((world[w][l][h].cost[up] == BLOCKED)
	|| (world[w][l][h].from == down)
```

```
        || (world[w][l][h+1].type == ADJ_TO_OBJECT)
        || (world[w][l][h+1].type == OBJECT))
                return(FALSE);

        dist += world[w][l][h].cost[up];

        if (world[w][l][h+1].type == GOAL)
                {
                if ((world[w][l][h+1].from == clear)
                || (dist < world[w][l][h+1].dist))
                        {
                        trim = dist;
                        world[w][l][h+1].dist = trim;
                        world[w][l][h+1].from = down;
                        return(TRUE);
                        }
                else
                        return(FALSE);
                }

        h++;
        if ((world[w][l][h].type == CLEAR)
        && ((world[w][l][h].from == clear)
        || (dist < world[w][l][h].dist)))
                {
                new=build_node(w, l, h, down, dist);
                if (new != (NODE*)NULL)
                        insert_node(new);
                }
        return(FALSE);
        }


void dump_world()
        {
        int a,b,c, hor,vert;

        for(c=0, hor=1; c<HEIGHT; c++, hor+=25)
                {
                gotoxy(hor,1);
                printf("time=t+%d\n", c);
                for(b=0, vert=2;b<LENGTH;b++, vert++)
                        {
                        gotoxy(hor, vert);
                        for(a=0;a<WIDTH;a++)
                                {
                                switch(world[a][b][c].type)
                                        {
                                        case GOAL:
                                        case START:
                                        case ADJ_TO_OBJECT:
                                        case OBJECT:
                                                printf("%c", world[a][b][c].type);
                                                break;
                                        case CLEAR:
                                                switch(world[a][b][c].from)
                                                        {
                                                        case right:
                                                                printf("R");
                                                                break;
                                                        case left:
                                                                printf("L");
                                                                break;
                                                        case up:
                                                                printf("U");
                                                                break;
                                                        case down:
                                                                printf("D");
                                                                break;
                                                        case front:
                                                                printf("F");
                                                                break;
                                                        case back:
                                                                printf("B");
                                                                break;
                                                        case backleft:
                                                                printf("T");
                                                                break;
                                                        case backright:
                                                                printf("U");
                                                                break;
                                                        case frontleft:
                                                                printf("V");
```

```
                                        break;
                                case frontright:
                                        printf("W");
                                        break;
                                case clear:
                                default:
                                        printf(".");
                                        break;
                                }
                        break;
                default:
                        printf("%c", world[a][b][c].type);
                        break;

                }
            printf("\n");
            }
        printf("\n\n");
        }
    }


int find_path()
    {
    NODE *cur;
    int result=FALSE, goal_found=FALSE;

    clrscr();
    timer_init();
    timer_start();
    while (head != (NODE*)NULL)
        {
        timer_stop();
        dump_world();
        timer_start();
        cur = head;
        result = check_line(cur);
        if (result)
            {
            goal_found = TRUE;
            mark_path();
            del_all_nodes();
            continue;
            }
        cur = head;  /* ripple out from the new head! */
        result = check_front(cur);
        if (result)
            {
            goal_found = TRUE;
            mark_path();
            del_all_nodes();
            continue;
            }
        result = check_frontleft(cur);
        if (result)
            {
            goal_found = TRUE;
            mark_path();
            del_all_nodes();
            continue;
            }
        result = check_left(cur);
        if (result)
            {
            goal_found = TRUE;
            mark_path();
            del_all_nodes();
            continue;
            }
        result = check_backleft(cur);
        if (result)
            {
            goal_found = TRUE;
            mark_path();
            del_all_nodes();
            continue;
            }
        result = check_back(cur);
        if (result)
            {
            goal_found = TRUE;
            mark_path();
            del_all_nodes();
            continue;
```

```
                }
            result = check_backright(cur);
            if (result)
                {
                goal_found = TRUE;
                mark_path();
                del_all_nodes();
                continue;
                }
            result = check_right(cur);
            if (result)
                {
                goal_found = TRUE;
                mark_path();
                del_all_nodes();
                continue;
                }
            result = check_frontright(cur);
            if (result)
                {
                goal_found = TRUE;
                mark_path();
                del_all_nodes();
                continue;
                }
            result = check_up(cur);
            if (result)
                {
                goal_found = TRUE;
                mark_path();
                del_all_nodes();
                continue;
                }
            del_node(cur);
            }
        timer_stop();
        dump_world();
        return(goal_found);
        }


void mark_path()
        {
        static char path='0';
        int flag=FALSE;
        int i, w, h, l, dist;

        w = end.w;
        l = end.l;
        for(h=0;(h<HEIGHT) && (world[w][l][h].dist==0.0);h++)
            ;
        end.h = h;
        dist = world[w][l][h].dist;
        for(i=dist; i>0; i--)
            {
            switch(world[w][l][h].from)
                {
                case front:
                    l++;
                    break;
                case right:
                    w++;
                    break;
                case left:
                    w--;
                    break;
                case back:
                    l--;
                    break;
                case frontright:
                    l++;
                    w++;
                    break;
                case frontleft:
                    l++;
                    w--;
                    break;
                case backright:
                    l--;
                    w++;
                    break;
                case backleft:
                    l--;
```

```
                                w--;
                                break;
                        case down:
                                h--;
                                break;
                        default:
                                break;
                        }
                if ((world[w][1][h].type != START)
                && ((world[w][1][h].type < '0')
                || (world[w][1][h].type > '9')))
                        {
                        world[w][1][h].type = path;
                        flag = TRUE;
                        }
                }
        if (flag)
                {
                path++;
                if (path > '9')
                        path = '0';
                }
        }
```

B.3. Standard Breadth First Graph Theory

The Standard Breadth First Search Algorithm for Graph Theory which was implemented as part of this project was coded in ANSI C++. The filenames for each of the separate source code files were supplied inside C++ format comments (i.e. //) at the beginning of each file listing. A detailed explanation of the design behind this program was presented in Chapter 4.

```
// main.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <sys\timeb.h>
#include <dir.h>
#include <ctype.h>
#include "edge.hpp"
#include "vertex.hpp"
#include "object.hpp"
#include "domain.hpp"
#include "bench.hpp"

#define MAPFILE_MASK "MAP*.DAT"

int main(int, char**);
void BuildVertexListFromDomain(VertexList&, Domain&);
int CalcRobotDir(VERTEX_NODE*);
void FindAllEdges(VertexList&, Domain&);
void MemStatus(char*);
int SetWorkingDir(void);
void RenameMapFile(char*);


int main(int argc, char** argv)
        {
        char* shortName;
        struct ffblk mapfile;

        for(;;)
                {
                shortName = strstr(argv[0], "\\");
                if (shortName == (char*)NULL)
                        {
                        shortName = argv[0];
                        cout << "This is " << shortName << "\n";
                        break;
                        }
                else
                        argv[0] = shortName+1;
```

```
        }
 MemStatus("Free memory before mainloop in main: ");

 if (!SetWorkingDir())
        {
        cout << "Program exiting gracefully\n";
        return(1);
        }
 if (findfirst(MAPFILE_MASK, &mapfile, 0))
        {
        cout << "Program exiting gracefully - no map files found\n";
        return(1);
        }
 do
        {
        char tmp[256];
        int i;
        Benchmark stopWatch(shortName);

        sprintf(tmp, "Starting on %s: ", mapfile.ff_name);
        MemStatus(tmp);
        for (i=0; i<NUMTIMES; i++)
                {
                int flag, timeTaken;
                float distTravelled;
                Domain theWorld(10, 20, 20, mapfile.ff_name);

                timeTaken = 0;
                distTravelled = 0.0;
                stopWatch.IterStart(i, mapfile.ff_name);
                stopWatch.Click();
                for(;;)
                        {
                        VertexList vertList;

                        BuildVertexListFromDomain(vertList, theWorld);
                        FindAllEdges(vertList, theWorld);
                        stopWatch.Click();
                        theWorld.DrawDomain();
                        stopWatch.Click();
                        flag = vertList.FindPath();
                        stopWatch.Click();
                        if (flag == TRUE)
                                cout << "Path found the GOAL!\n";
                        else
                                cout << "No path found to GOAL!\n";

                        if (flag)
                                {
                                VERTEX_NODE*t1, *t2;

                                t1 = vertList.GetStartVertex();
                                if (t1 != (VERTEX_NODE*)NULL)
                                        {
                                        t2 = t1->pathTo;
                                        if (t2 != (VERTEX_NODE*)NULL)
                                                {
                                                if ((t1->w == t2->w)
                                                && (t1->l == t2->l)
                                                && (t2->nodeType == GOAL))
                                                        {
                                                        cout << "MADE IT TO THE GOAL!\n";
                                                        break;
                                                        }
                                                distTravelled += theWorld.MoveRobot(t1->t, t1->w, t1->l,

                t2->t, t2->w, t2->l);
                                                }
                                        }
                                }
                        theWorld.AdvanceTime();
                        timeTaken++;
                        }
                stopWatch.IterStop(timeTaken, distTravelled);
                }
        stopWatch.LogCalcs();
        RenameMapFile(mapfile.ff_name);
//      break;
        }
 while (!findnext(&mapfile));

 MemStatus("Free memory after mainloop in main: ");
 return(0);
```

```
	}

void BuildVertexListFromDomain(VertexList &vList, Domain &domain)
	{
	char type;
	int t=domain.GetDomainTimeSlices(), a;
	int w=domain.GetDomainWidth(), b;
	int l=domain.GetDomainLength(), c;
	VERTEX_NODE*newPtr;

	for(a=0; a<t; a++)
	  {
		for(b=0; b<w; b++)
			{
			for(c=0; c<l; c++)
				{
				type = domain.GetPointType(a, b, c);
				if ((type == VERTEX)
				|| (type == START)
				|| (type == GOAL))
					{
					newPtr = vList.BuildNewVertex(a, b, c, type);
					vList.InsertNewVertex(newPtr);
					}
				}
			}
		}
	}

void FindAllEdges(VertexList &vList, Domain&domain)
	{
	EDGE_NODE*e;
	VERTEX_NODE*a,*b;
	float dist;

	for(a = vList.GetFirstVertex();
		a != (VERTEX_NODE*)NULL;
		a = vList.GetNextVertex(a))
		{
		for(b = vList.GetNextVertex(a);
			b != (VERTEX_NODE*)NULL;
			b = vList.GetNextVertex(b))
	  {
	  if (a == b)
				continue;

			if ((a->t == b->t)
			&& (a->w == b->w)
			&& (a->l == b->l))
				continue;

	  dist = domain.CheckLine(a->t, a->w, a->l, b->t, b->w, b->l);
	  if (dist > 0.0)
	     {
				e = a->edgeList->BuildNewEdge(b->t, b->w, b->l, dist);
				a->edgeList->InsertNewEdge(e);
				}
			dist = domain.CheckLine(b->t, b->w, b->l, a->t, a->w, a->l);
			if (dist > 0.0)
				{
				e = b->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
				b->edgeList->InsertNewEdge(e);
	     }
	  }
		}
	}

void MemStatus(char *StatusMessage)
	{
	char tmp[256];
	fstream debugFile;
	long MemLeft;

	debugFile.open("DEBUG.LOG", ios::app);
	if (!debugFile)
		cout << "Unable to open DEBUG.LOG\n";

	MemLeft = (long) coreleft();
	sprintf(tmp, "%s%ld\n", StatusMessage, MemLeft);
	debugFile.write(tmp, strlen(tmp));
	debugFile.close();
```

```
//      cout << StatusMessage << MemLeft << "\n";
        }


int SetWorkingDir()
        {
        char mapdir[256];

        cout <<"Enter the directory containing map files, or \"q\" for quit:";
        cin >> mapdir;
        if (toupper(mapdir[0]) == 'Q')
                {
                cout << "Quitting...\n";
                return(FALSE);
                }
        if (chdir(mapdir))
                {
                cout << "The directory " << mapdir << " could not be found.\n";
                return(FALSE);
                }
        cout << "Made " << mapdir << " the current directory.\n";
        return(TRUE);
        }


void RenameMapFile(char*filename)
        {
        char newfilename[128];
        char* ch;
        int i=(int)'.';

        strcpy(newfilename, filename);
        ch = strrchr(newfilename,i);
        if (ch != (char*)NULL)
                {
                strcpy(ch, ".bak");
                rename(filename, newfilename);
                }
        else
                exit(1);
        }

// bench.hpp
#define FALSE 0
#define TRUE !FALSE
#define NUMTIMES 10
#define MAXFILENAME 13

class Benchmark
        {
        public:
                Benchmark(char*);
                ~Benchmark();
                void Click(void);
                void IterStart(int, char*);
                void IterStop(int, float);
                void LogCalcs(void);
        private:
                void Diff(struct timeb*, struct timeb*, struct timeb*);

                fstream logFile;
                fstream avgFile;
                char mapFileName[MAXFILENAME];
                struct timeb benchmarks[NUMTIMES][2];
                float distRobotTravelled[NUMTIMES];
                int timeTaken[NUMTIMES];
                int currentIter;
                int clickToggleFlag;
                struct timeb elapsedTime, computeTime;
                struct timeb startTime, clickOnTime, clickOffTime;
        };

// bench.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <alloc.h>              // for coreleft()
#include <stdlib.h>             // for itoa()
#include <string.h>             // for strcpy()
#include <sys\timeb.h>
#include "bench.hpp"


Benchmark::Benchmark(char *fileName)
```

```
        {
        char *tmp;
        char logname[MAXFILENAME];

        while((tmp = strchr(fileName, '\\')) != (char*)NULL)
                fileName = tmp+1;
        strcpy(logname, fileName);
        if ((tmp = strchr(logname, '.')) != (char*)NULL)
                strcpy(tmp, ".LOG");
        else
                strcat(logname, ".LOG");
        logFile.open(logname, ios::app);
        if (!logFile)
                cout << "Unable to open " << logname << "\n";

        if ((tmp = strchr(logname, '.')) != (char*)NULL)
                strcpy(tmp, ".AVG");
        else
                strcat(logname, ".AVG");
        avgFile.open(logname, ios::app);
        if (!avgFile)
                cout << "Unable to open " << logname << "\n";

        cout << "Initialised Benchmark class\n";
        }


Benchmark::~Benchmark()
        {
        if (logFile)
                {
                logFile.flush();
                logFile.close();
                }
        if (avgFile)
                {
                avgFile.flush();
                avgFile.close();
                }
        cout << "Closed log files and Destroying Benchmark class\n";
        }


void Benchmark::Click()
        {
        switch(clickToggleFlag)
                {
                case TRUE:
                        ftime(&clickOnTime);
                        clickToggleFlag = FALSE;
                        break;
                case FALSE:
                default:
                        ftime(&clickOffTime);
                        Diff(&clickOnTime, &clickOffTime, &computeTime);
                        clickToggleFlag = TRUE;
                }
        }


void Benchmark::Diff(struct timeb*start, struct timeb*stop, struct timeb*diff)
        {
        if ((*stop).millitm < (*start).millitm)
                {
                (*stop).millitm += (short)1000; /* carry when subtracting, stops*/
                (*start).time += 1L;            /* negative wraparound problems!*/
                }
        (*diff).millitm += (*stop).millitm - (*start).millitm;
        (*diff).time += (long)((*diff).millitm / (short)1000);
        (*diff).millitm %= (short)1000;
        (*diff).time += ((*stop).time-(*start).time);
        }


void Benchmark::IterStart(int i, char*s)
        {
        currentIter = i;
        clickToggleFlag = TRUE;
        computeTime.time = elapsedTime.time = 0L;
        computeTime.millitm = elapsedTime.millitm = 0;

        strcpy(mapFileName, s);

        ftime(&startTime);
```

```
}


void Benchmark::IterStop(int t, float distTravelled)
	{
	struct timeb stopTime;

	ftime(&stopTime);
	Diff(&startTime, &stopTime, &elapsedTime);

	benchmarks[currentIter][0].time = elapsedTime.time;
	benchmarks[currentIter][0].millitm = elapsedTime.millitm;
	benchmarks[currentIter][1].time = computeTime.time;
	benchmarks[currentIter][1].millitm = computeTime.millitm;
	timeTaken[currentIter] = t;
	distRobotTravelled[currentIter] = distTravelled;
	}



void Benchmark::LogCalcs()
	{
	char tmp[256];
	float avgDist;
	struct timeb avg;
	int i, avgTimeTaken;

	for(i=0, avg.time=0L, avg.millitm=0, avgDist=0.0, avgTimeTaken=0;
		i<NUMTIMES;
		i++)
		{
		sprintf(tmp, "%s (%02d) Elapsed time:%05ld.%03d\n", mapFileName, i,
					benchmarks[i][0].time, benchmarks[i][0].millitm);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Compute time:%05ld.%03d\n", mapFileName, i,
					benchmarks[i][1].time, benchmarks[i][1].millitm);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Dist travelled:%f\n", mapFileName, i,
					distRobotTravelled[i]);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Time Slices Taken:%d\n", mapFileName, i,
					timeTaken[i]);
		logFile.write(tmp, strlen(tmp));
		avg.time+=benchmarks[i][0].time;
		avg.millitm+=benchmarks[i][0].millitm;
		if (avg.millitm % 1000 != avg.millitm)
			{
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;
			}
		}

	sprintf(tmp, "%s Tot. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));
	cout << avg.time << "." << avg.millitm << "\n";
	cout << tmp << "\n";

	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;

	sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	avgFile.write(tmp, strlen(tmp));
//	sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));

	for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
		{
		avg.time+=benchmarks[i][1].time;
		avg.millitm+=benchmarks[i][1].millitm;
		if (avg.millitm % 1000 != avg.millitm)
			{
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;

		}
	sprintf(tmp, "%s Tot. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));
	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;
```

```
        sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));

        for(i=0, avgTimeTaken=0; i<NUMTIMES; i++)
                avgTimeTaken+=timeTaken[i];
        avgTimeTaken = avgTimeTaken / NUMTIMES;
        sprintf(tmp, "%s Avg. Time Taken:%d\n", mapFileName, avgTimeTaken);
        avgFile.write(tmp, strlen(tmp));
        logFile.write(tmp, strlen(tmp));

        for(i=0, avgDist=0.0; i<NUMTIMES; i++)
                avgDist += distRobotTravelled[i];
        avgDist = avgDist / ((float)NUMTIMES);
        sprintf(tmp, "%s Avg. Dist Travelled:%f\n", mapFileName, avgDist);
        avgFile.write(tmp, strlen(tmp));
        logFile.write(tmp, strlen(tmp));

        sprintf(tmp, "%s =====================\n", mapFileName);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s =====================\n", mapFileName);
        logFile.write(tmp, strlen(tmp));
        }

// domain.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00

#define FROM_FRONT 0
#define FROM_BACK 1
#define FROM_LEFT 2
#define FROM_RIGHT 3
#define FROM_FRONTRIGHT 4
#define FROM_FRONTLEFT 5
#define FROM_BACKRIGHT 6
#define FROM_BACKLEFT 7
#define FROM_UP 8
#define FROM_DOWN 9
#define FROM_NOWHERE 10

#define NOCOST (float)0.0
#define NORMAL (float)1.0
#define NORMAL_DIAG (float)1.414214
#define BLOCKED (float)1000.0

typedef struct
        {
        float dist;
        int from;
        char type;
        float cost[9];
        } POINT;

class Domain
        {
        public:
                Domain(int, int, int, char*);
                ~Domain();
                void AdvanceTime(void);
        float CheckLine(int, int, int, int, int, int);
                void DrawDomain(void);
        int GetDomainLength(void) { return(domainLength); }
        int GetDomainTimeSlices(void) { return(domainTimeSlices); }
        int GetDomainWidth(void) { return(domainWidth); }
                POINT* GetPoint(int, int, int);
        float GetPointCost(int, int, int, int);
                char GetPointFrom(int, int, int);
                char GetPointType(int, int, int);
                int IsPointClear(int, int, int);
```

```
        nt IsPointGoal(int, int, int);
        int IsPointNearObject(int, int, int);
        int IsPointObject(int, int, int);
        int IsPointStart(int, int, int);
int IsPointVertex(int, int, int);
        float MoveRobot(int, int, int, int, int, int);
        void SetPointFrom(int, int, int, int);
        void SetPointType(int, int, int, char);
 private:
        void AgeTimeSlices(void);
int CalcRobotDir(int, int, int, int, int, int);
        int ClearAdjPointOK(int, int, int);
        void ClearMobileObject(int, int, int);
        void ClearVerticesInTimeSlice(int);
        void DrawTimeSlice(int);
        void InitTimeSlice(int);
        void MarkMobileObject(int, int, int);
        void MoveMobileObject(OBJECT_NODE*);
        void MoveMobileObjects(void);
        void SetAdjObjsInTimeSlice(int);
        void SetGoalFromFile(char*);
        void SetMobileObjsFromFile(char*);
        void SetPermObjsInTimeSlice(int);
        void SetStartFromFile(char*);
        void SetVerticesInTimeSlice(int);

        ObjectList objList;
        POINT** domainHead;
int domainWidth;
int domainLength;
int domainTimeSlices;
 };

// domain.cpp
#include <iostream.h>
#include <fstream.h>
#include <alloc.h>          // for coreleft()
#include <stdio.h>
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <conio.h>
#include "object.hpp"
#include "domain.hpp"

Domain::Domain(int numTimeSlices, int width, int length, char*mapFileName)
        {
        int i;

        cout << "Contructing Domain class\n";
        domainTimeSlices = numTimeSlices;
        domainWidth = width;
        domainLength = length;

        cout << "FreeHeap:" << farcoreleft() << "\n";
        cout << "Amount needed for one timeslice:";
        cout << width*length*sizeof(POINT) << "\n";

        domainHead = (POINT**)farcalloc(numTimeSlices, sizeof(POINT**));
        if (domainHead == (POINT**)NULL)
                {
                cout << "Not enough memory to build model of world\n";
                domainHead = (POINT***)NULL;
                domainTimeSlices = domainWidth = domainLength = 0;
                return;
                }

        for(i=0; i<numTimeSlices; i++)
                {
                domainHead[i] = (POINT*)farcalloc(width*length, sizeof(POINT));
                cout << "FreeHeap after timeslice allocated:" << farcoreleft() << "\n";
                if (domainHead[i] == (POINT*)NULL)
                        {
                        if (i==0)
                                cout << "Not enough memory to build model of world\n";
                        else
                                cout << "Only enough memory to build " << i << " of the ";
                                cout << numTimeSlices << " timeslices in model of world\n";
                        domainTimeSlices = i;
                        break;
                        }
                InitTimeSlice(i);
                SetPermObjsInTimeSlice(i);
                }
        SetStartFromFile(mapFileName);
```

```
        SetGoalFromFile(mapFileName);
        SetMobileObjsFromFile(mapFileName);
        for(i=0; i<domainTimeSlices; i++)
                {
                SetAdjObjsInTimeSlice(i);
                SetVerticesInTimeSlice(i);
                }
        for(i=0; i<domainTimeSlices-1; i++)
                AdvanceTime();
        cout << "FreeHeap after Domain allocated:" << farcoreleft() << "\n";
        }


Domain::~Domain()
        {
        POINT *tmp;
        int i;

        cout << "Destructing Domain class\n";
        if (domainHead == (POINT**)NULL)
                {
                cout << "Not freeing domain - DomainHead was NULL\n";
                return;
                }
        for(i=0; i<domainTimeSlices; i++)
                {
                tmp = domainHead[i];
                farfree(tmp);
                }
        farfree(domainHead);
        domainHead = (POINT**)NULL;
        cout << "FreeHeap after Domain deallocated:" << farcoreleft() << "\n";
//      getch();
        }


void Domain::AdvanceTime()
        {
        AgeTimeSlices();
//      DrawDomain();
        ClearVerticesInTimeSlice(domainTimeSlices-1);
        MoveMobileObjects();
//      DrawDomain();
        SetVerticesInTimeSlice(domainTimeSlices-1);
//      DrawDomain();
        }


///////////////////////////////////////
//
// Copy the contents of every timeslice
// into the previous timeslice
// [0] = [1], [1] = [2], etc.
// Leave the last timeslice unchanged.
// Another routine will decide the moves
// for all the mobile objects.
//
// A quick way to do this is moving ptrs
// to the timeslices and only copying the
// contents of the last timeslice over the
// contents of the first timeslice
//
///////////////////////////////////////
void Domain::AgeTimeSlices()
        {
        int i, j;
        POINT *tmp;

        tmp = domainHead[0];
        for(i=0, j=1; i<domainTimeSlices-1; i++, j++)
                domainHead[i] = domainHead[j];
        domainHead[domainTimeSlices-1] = tmp;
        _fmemcpy(domainHead[domainTimeSlices-1],
                                domainHead[domainTimeSlices-2],
                                sizeof(*domainHead[domainTimeSlices-1]));
        }


int Domain::CalcRobotDir(int st, int sw, int sl, int et, int ew, int el)
   {
        if ((!IsPointStart(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(clear);
```

```
if ((ew - sw > 0)
&& (el - sl > 0))
        {
        sw++, sl++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(frontright);
        else
                return(clear);
        }

if ((ew - sw > 0)
&& (el - sl == 0))
        {
        sw++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(right);
        else
                return(clear);
        }

if ((ew - sw > 0)
&& (el - sl < 0))
        {
        sw++, sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(backright);
        else
                return(clear);
        }

if ((ew - sw == 0)
&& (el - sl > 0))
        {
        sl++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(front);
        else
                return(clear);
        }

if ((ew - sw == 0)  // silly, but just in case
&& (el - sl == 0))
        {
        return(clear);
        }

if ((ew - sw == 0)
&& (el - sl < 0))
        {
        sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(back);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl > 0))
        {
        sw--, sl++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(frontleft);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl == 0))
        {
        sw--;
        if ((IsPointClear(st, sw, sl))
```

```
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(left);
		else
			return(clear);
		}

	if ((ew - sw < 0)
	&& (el - sl < 0))
		{
		sw--, sl--;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(backleft);
		else
			return(clear);
		}
	return(clear);
	}


float Domain::CheckLine(int st, int sw, int sl, int et, int ew, int el)
	{
//	static int displayCounter=1;
	int tmpt, tmpw, tmpl;
	float dist;

	for(dist=0.0, tmpt=et-st, tmpw=ew-sw, tmpl=el-sl;
		(tmpt != 0) || (tmpw != 0) || (tmpl != 0);
		tmpt=et-st, tmpw=ew-sw, tmpl=el-sl)
		{
		if (tmpt < 0)
			{
			return(-1.0);
			}
		if ((tmpw > 0)
		&& (tmpl > 0))
			{
			dist += GetPointCost(st, sw++, sl++, frontright);
			if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
	if ((tmpw > 0)
		&& (tmpl == 0))
			{
	  dist += GetPointCost(st, sw++, sl, right);
		if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
			}
		if ((tmpw > 0)
		&& (tmpl < 0))
			{
	  dist += GetPointCost(st, sw++, sl--, backright);
			if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
		if ((tmpw < 0)
		&& (tmpl > 0))
			{
	  dist += GetPointCost(st, sw--, sl++, frontleft);
		if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
		if ((tmpw < 0)
		&& (tmpl == 0))
			{
	  dist += GetPointCost(st, sw--, sl, left);
		if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
		if ((tmpw < 0)
		&& (tmpl < 0))
```

```
                {
        dist += GetPointCost(st, sw--, sl--, backleft);
                if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
        continue;
                }
            if ((tmpw == 0)
            && (tmpl > 0))
                {
        dist += GetPointCost(st, sw, sl++, front);
            if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
        continue;
        }
            if ((tmpw == 0)
            && (tmpl < 0))
                {
        dist += GetPointCost(st, sw, sl--, back);
            if ((!IsPointClear(st, sw, sl))
            && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
        continue;
        }
      if ((tmpw == 0)
      && (tmpl == 0)
      && (tmpt >0))
                {
        dist += GetPointCost(st++, sw, sl, up);
            if ((!IsPointClear(st, sw, sl))
            && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
        continue;
        }
            }
//      cout << "CheckLine:" << displayCounter++ << ":returned" << dist << "\n";
        return(dist);
        }


int Domain::ClearAdjPointOK(int t, int w, int l)
        {
        char type;
        int a,b;

        for(a=w-1;a<=w+1;a++)
                {
                if ((a < 0) || (a >= domainWidth))
                        continue;
                for(b=l-1;b<=l+1;b++)
                        {
                        if ((b < 0) || (b >= domainLength))
                                continue;
                        if ((a == w) && (b == l))
                                continue;

          type = GetPointType(t, a, b);
                        if ((type == OBJECT)
                        ||  (type == MOBILE_OBJECT))
                                return(FALSE);
                        }
                }
        return(TRUE);
        }


void Domain::ClearMobileObject(int t, int w, int l)
   {
        int i, j;

        if (GetPointType(t, w, l) == MOBILE_OBJECT)
                {
                if (ClearAdjPointOK(t, w, l))
                        {
                        SetPointType(t, w, l, CLEAR);
                        SetPointFrom(t, w, l, FROM_NOWHERE);
                        }
                else
                        {
                        SetPointType(t, w, l, ADJ_TO_OBJECT);
                        SetPointFrom(t, w, l, FROM_NOWHERE);
                        }
                }
```

```
         for(i=w-1, j=l-1; j<l+2;)
                 {
        if ((GetPointType(t, i, j) == ADJ_TO_OBJECT)
                 && (ClearAdjPointOK(t, i, j)))
                 {
                 SetPointType(t, i, j, CLEAR);
                         SetPointFrom(t, i, j, FROM_NOWHERE);
                         }
                 if (i == w+1)
                         {
                         i = w-1;
                         j++;
                         }
                 else
                         i++;
                 }
         }


void Domain::ClearVerticesInTimeSlice(int t)
         {
         int w, l;

         for(w=0; w<domainWidth; w++)
                 {
                 for(l=0; l<domainLength; l++)
                         {
                         if (GetPointType(t, w, l) == VERTEX)
                 {
                                 SetPointFrom(t, w, l, FROM_NOWHERE);
                                 SetPointType(t, w, l, CLEAR);
                 continue;
                 }
                         if (GetPointType(t, w, l) == MOBILE_OBJECT)
                                 {
                 ClearMobileObject(t, w, l);
                                 }
                         }
                 }
         }


void Domain::DrawDomain()
         {
         int i;

         for (i=0; i< domainTimeSlices; i++)
                 {
                 DrawTimeSlice(i);
//               getch();
                 }
         }


void Domain::DrawTimeSlice(int timeSlice)
         {
         char type;
         int a,b,y;

         clrscr();
         gotoxy(1,1);
         cout << "time=t+" << timeSlice;

         for(b=0, y=2;b<domainLength; b++, y++)
                 {
                 gotoxy(1, y);
                 for(a=0; a<domainWidth; a++)
                         {
                         type = GetPointType(timeSlice, a, b);
                         switch(type)
                                 {
                                 case GOAL:
                                 case START:
                                 case ADJ_TO_OBJECT:
                                 case OBJECT:
                                 case MOBILE_OBJECT:
                                 case VERTEX:
                                         cout << type;
                                         break;
                                 case CLEAR:
                                         switch(GetPointFrom(timeSlice, a, b))
                                                 {
                                                 case FROM_RIGHT:
                                                         cout << "R";
```

```
                                        break;
                                case FROM_LEFT:
                                        cout << "L";
                                        break;
                                case FROM_UP:
                                        cout << "U";
                                        break;
                                case FROM_DOWN:
                                        cout << "D";
                                        break;
                                case FROM_FRONT:
                                        cout << "F";
                                        break;
                                case FROM_BACK:
                                        cout << "B";
                                        break;
                                case FROM_BACKLEFT:
                                        cout << "T";
                                        break;
                                case FROM_BACKRIGHT:
                                        cout << "U";
                                        break;
                                case FROM_FRONTLEFT:
                                        cout << "V";
                                        break;
                                case FROM_FRONTRIGHT:
                                        cout << "W";
                                        break;
                                case FROM_NOWHERE:
                                default:
                                        cout << ".";
                                        break;
                                }
                        break;
                default:
                        cout << "?";
                        break;
                }
        }
        cout << "\n";
        }
    }

POINT* Domain::GetPoint(int timeSlice, int width, int length)
        {
    POINT *tmp, *tmp2;

    tmp = domainHead[timeSlice];
        tmp2 = tmp + (domainWidth*width) + length;
    return(tmp2);
    }


float Domain::GetPointCost(int timeSlice, int width, int length, int dir)
    {
    POINT *tmp;

    tmp = GetPoint(timeSlice, width, length);
        return(tmp->cost[dir]);
    }

char Domain::GetPointFrom(int timeSlice, int width, int length)
    {
    POINT *tmp;

    tmp = GetPoint(timeSlice, width, length);
        return(tmp->from);
    }


char Domain::GetPointType(int timeSlice, int width, int length)
        {
    POINT *tmp;

        tmp = GetPoint(timeSlice, width, length);
        return(tmp->type);
        }


void Domain::InitTimeSlice(int timeSlice)
        {
        POINT*tmp;
    int a,b;
```

```
for(a=0;a<domainWidth;a++)
            {
            for(b=0;b<domainLength;b++)
                    {
       tmp = GetPoint(timeSlice, a, b);
                    tmp->dist=0.0;
                    tmp->from=FROM_NOWHERE;
                    tmp->type=CLEAR;

                    if (timeSlice == domainTimeSlices-1)
                           tmp->cost[FROM_UP] = BLOCKED;
                    else
                           tmp->cost[FROM_UP] = NORMAL;

                    if ((a == 0) && (b == 0))
                           {
                           tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                           tmp->cost[FROM_RIGHT] = NORMAL;
                           tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                           tmp->cost[FROM_BACK] = BLOCKED;
                           tmp->cost[FROM_BACKLEFT] = BLOCKED;
                           tmp->cost[FROM_LEFT] = BLOCKED;
                           tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                           tmp->cost[FROM_FRONT] = NORMAL;
                           continue;
                           }
                    if ((a == 0) && (b < domainLength-1))
                           {
                           tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                           tmp->cost[FROM_RIGHT] = NORMAL;
                           tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                           tmp->cost[FROM_BACK] = NORMAL;
                           tmp->cost[FROM_BACKLEFT] = BLOCKED;
                           tmp->cost[FROM_LEFT] = BLOCKED;
                           tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                           tmp->cost[FROM_FRONT] = NORMAL;
                           continue;
                           }
                    if ((a == 0) && (b==domainLength-1))
                           {
                           tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                           tmp->cost[FROM_RIGHT] = NORMAL;
                           tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                           tmp->cost[FROM_BACK] = NORMAL;
                           tmp->cost[FROM_BACKLEFT] = BLOCKED;
                           tmp->cost[FROM_LEFT] = BLOCKED;
                           tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                           tmp->cost[FROM_FRONT] = BLOCKED;
                           continue;
                           }
                    if ((a < domainWidth-1) && (b == 0))
                           {
                           tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                           tmp->cost[FROM_RIGHT] = NORMAL;
                           tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                           tmp->cost[FROM_BACK] = BLOCKED;
                           tmp->cost[FROM_BACKLEFT] = BLOCKED;
                           tmp->cost[FROM_LEFT] = NORMAL;
                           tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                           tmp->cost[FROM_FRONT] = NORMAL;
                           continue;
                           }
                    if ((a < domainLength-1) && (b < domainWidth-1))
                           {
                           tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                           tmp->cost[FROM_RIGHT] = NORMAL;
                           tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                           tmp->cost[FROM_BACK] = NORMAL;
                           tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                           tmp->cost[FROM_LEFT] = NORMAL;
                           tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                           tmp->cost[FROM_FRONT] = NORMAL;
                           continue;
                           }
                    if ((a < domainWidth-1) && (b == domainLength-1))
                           {
                           tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                           tmp->cost[FROM_RIGHT] = NORMAL;
                           tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                           tmp->cost[FROM_BACK] = NORMAL;
                           tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                           tmp->cost[FROM_LEFT] = NORMAL;
                           tmp->cost[FROM_FRONTLEFT] = BLOCKED;
```

```
                                        tmp->cost[FROM_FRONT] = BLOCKED;
                                        continue;
                                        }
                                if ((a == domainWidth-1) && (b == 0))
                                        {
                                        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                                        tmp->cost[FROM_RIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACK] = BLOCKED;
                                        tmp->cost[FROM_BACKLEFT] = BLOCKED;
                                        tmp->cost[FROM_LEFT] = NORMAL;
                                        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                                        tmp->cost[FROM_FRONT] = NORMAL;
                                        continue;
                                        }
                                if ((a == domainWidth-1) && (b < domainWidth-1))
                                        {
                                        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                                        tmp->cost[FROM_RIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACK] = NORMAL;
                                        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                                        tmp->cost[FROM_LEFT] = NORMAL;
                                        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                                        tmp->cost[FROM_FRONT] = NORMAL;
                                        continue;
                                        }
                                if ((a == domainLength-1) && (b == domainWidth-1))
                                        {
                                        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                                        tmp->cost[FROM_RIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACK] = NORMAL;
                                        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                                        tmp->cost[FROM_LEFT] = NORMAL;
                                        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                                        tmp->cost[FROM_FRONT] = BLOCKED;
                                        continue;
                                        }
                                }
                        }
                }


int Domain::IsPointClear(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
   || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case CLEAR:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointGoal(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
   || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case GOAL:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }
```

```
int Domain::IsPointNearObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
                case OBJECT:
                        return(OBJECT);
                case MOBILE_OBJECT:
                        return(MOBILE_OBJECT);
                case ADJ_TO_OBJECT:
                        return(ADJ_TO_OBJECT);
                default:
                        return(FALSE);
                }
    }

int Domain::IsPointObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
                case OBJECT:
                        return(OBJECT);
                case MOBILE_OBJECT:
                        return(MOBILE_OBJECT);
                case ADJ_TO_OBJECT:
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointStart(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case START:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointVertex(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case VERTEX:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }
```

```
void Domain::MarkMobileObject(int t, int w, int l)
        {
        int i, j;

        if ((GetPointType(t, w, l) == CLEAR)
        || (GetPointType(t, w, l) == ADJ_TO_OBJECT))
                {
                SetPointType(t, w, l, MOBILE_OBJECT);
                SetPointFrom(t, w, l, FROM_NOWHERE);
                }
        for(i=w-1, j=l-1; j<l+2;)
                {
        if (GetPointType(t, i, j) == CLEAR)
                {
                SetPointType(t, i, j, ADJ_TO_OBJECT);
                SetPointFrom(t, i, j, FROM_NOWHERE);
                }
        if (i == w+1)
          {
          i = w-1;
          j++;
          }
        else
                        i++;
                }
        }


void Domain::MoveMobileObject(OBJECT_NODE*object)
        {
        int t=domainTimeSlices-1, w=(*object).w, l=(*object).l;

        ClearMobileObject(t, w, l);
        switch((*object).direction)
                {
                case front:
                        if (IsPointObject(t, w, l+1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).l++;
                        break;
                case frontleft:
                        if (IsPointObject(t, w-1, l+1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).l++, (*object).w--;
                        break;
                case left:
                        if (IsPointObject(t, w-1, l))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w--;
                        break;
                case backleft:
                        if (IsPointObject(t, w-1, l-1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w--, (*object).l--;
                        break;
                case back:
                        if (IsPointObject(t, w, l-1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).l--;
                        break;
                case backright:
                        if (IsPointObject(t, w+1, l-1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w++, (*object).l--;
                        break;
                case right:
                        if (IsPointObject(t, w+1, l))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w++;
                        break;
                case frontright:
                        if (IsPointObject(t, w+1, l+1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w++, (*object).l++;
                        break;
```

```
		default:
			cout << "*** unknown direction for mobile object ignored ***\n";
			break;
		}
	MarkMobileObject(t, object->w, object->l);
	}


void Domain::MoveMobileObjects()
	{
	OBJECT_NODE*cur, *orig;

	orig = cur = objList.GetNextObject();
	if (cur == (OBJECT_NODE*)NULL)
	return;

	do
		{
		if ((*cur).velocity > 0)
			MoveMobileObject(cur);

	cur = objList.GetNextObject();
		} while (cur != orig);
	}


float Domain::MoveRobot(int st, int sw, int sl, int et, int ew, int el)
	{
	float dist = 0.0;
	int dir;

	if (!IsPointStart(st, sw, sl))
		return(dist);

	dir = CalcRobotDir(st, sw, sl, et, ew, el);
	SetPointType(st, sw, sl, CLEAR);
	SetPointFrom(st, sw, sl, FROM_NOWHERE);

	switch(dir)
		{
		case frontright:
			dist = GetPointCost(st, sw, sl, frontright);
			st++, sw++, sl++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw--, sl--;
				dist = 0.0;
				}
			break;
		case right:
			dist = GetPointCost(st, sw, sl, right);
			st++, sw++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw--;
				dist = 0.0;
				}
			break;
		case backright:
			dist = GetPointCost(st, sw, sl, backright);
			st++, sw++, sl--;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw--, sl++;
				dist = 0.0;
				}
			break;
		case frontleft:
			dist = GetPointCost(st, sw, sl, frontleft);
			st++, sw--, sl++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw++, sl--;
				dist = 0.0;
				}
```

```
				break;
			case left:
				dist = GetPointCost(st, sw, sl, left);
				st++, sw--;
				if ((!IsPointClear(st, sw, sl))
				&& (!IsPointVertex(st, sw, sl))
				&& (!IsPointGoal(st, sw, sl)))
					{
					st--, sw++;
					dist = 0.0;
					}
				break;
			case backleft:
				dist = GetPointCost(st, sw, sl, backleft);
				st++, sw--, sl--;
				if ((!IsPointClear(st, sw, sl))
				&& (!IsPointVertex(st, sw, sl))
				&& (!IsPointGoal(st, sw, sl)))
					{
					st--, sw++, sl++;
					dist = 0.0;
					}
				break;
			case front:
				dist = GetPointCost(st, sw, sl, front);
				st++, sl++;
				if ((!IsPointClear(st, sw, sl))
				&& (!IsPointVertex(st, sw, sl))
				&& (!IsPointGoal(st, sw, sl)))
					{
					st--, sl--;
					dist = 0.0;
					}
				break;
			case back:
				dist = GetPointCost(st, sw, sl, back);
				st++, sl--;
				if ((!IsPointClear(st, sw, sl))
				&& (!IsPointVertex(st, sw, sl))
				&& (!IsPointGoal(st, sw, sl)))
					{
					st--, sl++;
					dist = 0.0;
					}
				break;
			case up:
				dist = GetPointCost(st, sw, sl, up);
				st++;
				if ((!IsPointClear(st, sw, sl))
				&& (!IsPointVertex(st, sw, sl))
				&& (!IsPointGoal(st, sw, sl)))
					{
					st--;
					dist = 0.0;
					}
				break;
			default:
				break;
			}
		SetPointType(st, sw, sl, START);
		SetPointFrom(st, sw, sl, FROM_NOWHERE);
		return(dist);
		}


void Domain::SetAdjObjsInTimeSlice(int timeSlice)
		{
		int a,b;
		int w,l;

		for(w=0; w<domainWidth; w++)
			{
			for(l=0; l<domainLength; l++)
				{
				if (!IsPointObject(timeSlice, w, l))
					continue;

				for(a=w-1; a<=w+1; a++)
					{
					if ((a < 0) || (a >= domainWidth))
						continue;
					for(b=l-1; b<=l+1; b++)
						{
						if ((b < 0) || (b >= domainLength))
```

```
                                        continue;

                                if (GetPointType(timeSlice, a, b) == CLEAR)
                                        {
                                        SetPointFrom(timeSlice, a, b, FROM_NOWHERE);
                                        SetPointType(timeSlice, a, b, ADJ_TO_OBJECT);
                                        }

                        }
                }
            }
        }


void Domain::SetGoalFromFile(char*fileName)
        {
        int i, w, l;
        char recType;
        char tmp[256];
        fstream dataFile;

        dataFile.open(fileName, ios::in);
        if (!dataFile)
                cout << "Unable to open " << fileName << ". No Goal added.\n";

        for(;;)
                {
                dataFile. getline(tmp, sizeof(tmp));
                if (strlen(tmp) == 0)
                        break;
                if (tmp[0] == GOAL)
                        {
//                      cout << "The GOAL line is: " << tmp << "\n";
                        if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                                {
            for(i=0; i<domainTimeSlices; i++)
              {
                                SetPointFrom(i, w, l, FROM_NOWHERE);
                                SetPointType(i, w, l, GOAL);
              }
                                break;
                                }
                        else
                                cout << "Improperly formatted line ignored\n";
                        }
                }
        dataFile.close();
        }


void Domain::SetMobileObjsFromFile(char*fileName)
        {
        int w, l;
        char recType;
        char tmp[256];
        fstream dataFile;
        OBJECT_NODE* newObj;

        dataFile.open(fileName, ios::in);
        if (!dataFile)
                cout << "Unable to open " << fileName << ". No Mobile objs added.\n";

        for(;;)
                {
                int i;

                dataFile. getline(tmp, sizeof(tmp));
                i = strlen(tmp);
                if (i <= 0)
                        break;
                cout << "Length of input line is: " << i << "\n";
                if ((tmp[0] != GOAL)
                && (tmp[0] != START))
                        {
//                      cout << "OBJECT line is: " << tmp << "\n";
                        if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                                {
                                SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
                                SetPointType(domainTimeSlices-1, w, l, MOBILE_OBJECT);
                                newObj = objList.BuildNewObject(w, l);
                                if (newObj)
                                        {
                                        objList.InsertNewObject(newObj);
//                                      cout << "Added obj to obj list\n";
```

```
                                }
                        else
                                cout << "Improperly formatted line ignored\n";
                        }
                }
        dataFile.close();
        }


void Domain::SetPermObjsInTimeSlice(int timeSlice)
        {
        int w, l;

        for(w=0; w<domainWidth-5; w++)
                {
                l=2;
                switch(w)
                        {
                        case 3:
                        case 4:
                        case 5:
                                break;
                        default:
                                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                                SetPointType(timeSlice, w, l, OBJECT);
//                              SetAdjObjsInTimeSlice(timeSlice, w, l);
                                break;
                        }
                }
        for(w=0; w<domainWidth-10; w++)
                {
                l=7;
                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                SetPointType(timeSlice, w, l, OBJECT);
//              SetAdjObjsInTimeSlice(timeSlice, w, l);
                }
        for(l=0, w=domainWidth-10; l<domainLength-2; l++)
                {
                switch(l)
                        {
                        case 3:
                        case 4:
                        case 5:
                        case 6:
                        case 14:
                        case 15:
                        case 16:
                                break;
                        default:
                                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                                SetPointType(timeSlice, w, l, OBJECT);
//                              SetAdjObjsInTimeSlice(timeSlice, w, l);
                                break;
                        }
                }
        for(l=2, w=domainWidth-5; l<domainLength-7; l++)
                {
                switch(l)
                        {
                        case 9:
                        case 10:
                        case 11:
                                break;
                        default:
                                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                                SetPointType(timeSlice, w, l, OBJECT);
//                              SetAdjObjsInTimeSlice(timeSlice, w, l);
                                break;
                        }
                }
        for(w=domainWidth-5, l=domainLength-7; w<domainWidth; w++)
                {
                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                SetPointType(timeSlice, w, l, OBJECT);
//              SetAdjObjsInTimeSlice(timeSlice, w, l);
                }
        for(w=domainWidth-10, l=domainLength-2; w<domainWidth; w++)
                {
                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                SetPointType(timeSlice, w, l, OBJECT);
//              SetAdjObjsInTimeSlice(timeSlice, w, l);
                }
        }
```

```
void Domain::SetPointFrom(int timeSlice, int width, int length, int from)
	{
	POINT *tmp;

	if ((width < 0)
	|| (length < 0)
	|| (timeSlice < 0)
	|| (width >= domainWidth)
	|| (length >= domainLength)
	|| (timeSlice >= domainTimeSlices))
		return;
	tmp = GetPoint(timeSlice, width, length);
	tmp->from = from;
	}

void Domain::SetPointType(int timeSlice, int width, int length, char type)
	{
	char oldType;
	POINT *tmp;

	if ((width < 0)
	|| (length < 0)
	|| (timeSlice < 0)
	|| (width >= domainWidth)
	|| (length >= domainLength)
	|| (timeSlice >= domainTimeSlices))
		return;
	tmp = GetPoint(timeSlice, width, length);
	oldType = tmp->type;
	tmp->type = type;
	if (oldType != OBJECT)
		{
//		SetAdjObjsInTimeSlice(timeSlice, width, length);
		}
	}


void Domain::SetStartFromFile(char*fileName)
	{
	int w, l;
	char recType;
	char tmp[256];
	fstream dataFile;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Start added.\n";

	for(;;)
		{
		dataFile. getline(tmp, sizeof(tmp));
		if (strlen(tmp) == 0)
			break;
		if (tmp[0] == START)
			{
//			cout << "The START line is: " << tmp << "\n";
			if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
				{
				SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
				SetPointType(domainTimeSlices-1, w, l, START);
				break;
				}
			else
				cout << "Improperly formatted line ignored\n";
			}
		}
	dataFile.close();
	}


void Domain::SetVerticesInTimeSlice(int t)
	{
	int w, l;
	int diff_counter, corner_counter;
	int side_bits, corner_bits;

	for(w=0; w<domainWidth; w++)
		{
		for(l=0; l<domainLength; l++)
			{
			if (!IsPointClear(t, w, l))
				continue;
```

```
			corner_bits = BITMASK_CLEAR;
			corner_counter = 0;
			if (IsPointNearObject(t, w-1, l-1))
				{
				corner_bits |= BITMASK_LEFT;
				corner_bits |= BITMASK_TOP;
				corner_counter++;
				}
			if (IsPointNearObject(t, w-1, l+1))
				{
				corner_bits |= BITMASK_LEFT;
				corner_bits |= BITMASK_BOTTOM;
				corner_counter++;
				}
			if (IsPointNearObject(t, w+1, l+1))
				{
				corner_bits |= BITMASK_RIGHT;
				corner_bits |= BITMASK_BOTTOM;
				corner_counter++;
				}
			if (IsPointNearObject(t, w+1, l-1))
				{
				corner_bits |= BITMASK_RIGHT;
				corner_bits |= BITMASK_TOP;
				corner_counter++;
				}
			if (corner_bits != BITMASK_CLEAR)
				{
				side_bits = BITMASK_CLEAR;
				if (IsPointNearObject(t, w-1, l))
					side_bits |= BITMASK_LEFT;
				if (IsPointNearObject(t, w, l-1))
					side_bits |= BITMASK_TOP;
				if (IsPointNearObject(t, w+1, l))
					side_bits |= BITMASK_RIGHT;
				if (IsPointNearObject(t, w, l+1))
					side_bits |= BITMASK_BOTTOM;

				for(diff_counter=0;
					(side_bits != BITMASK_CLEAR)
					|| (corner_bits != BITMASK_CLEAR);
					side_bits >>= 1, corner_bits >>= 1)
					{
					if ((corner_bits & (int)0x01)
					&& ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
						{
						diff_counter++;
						continue;
						}
					if ((side_bits & (int)0x01)
					&& ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
						{
						diff_counter++;
						continue;
						}
					}
				if ((diff_counter > 2)
				|| ((diff_counter == 2) && (corner_counter == 1)))
					{
					SetPointFrom(t, w, l, FROM_NOWHERE);
					SetPointType(t, w, l, VERTEX);

				}
			}
		}
	}

// edge.hpp
typedef struct van
	{
	int t,w,l;
	float dist;
	struct van *prev, *next;
	} EDGE_NODE;

class EdgeList
	{
	public:
		EdgeList();
		~EdgeList();
	EDGE_NODE* BuildNewEdge(int, int, int, float);
	void DelEdge(EDGE_NODE*);
		void DelEdgeToVertex(int, int, int);
```

```
        EDGE_NODE* GetFirstEdge(void);
        EDGE_NODE* GetNextEdge(EDGE_NODE*);
        void InsertNewEdge(EDGE_NODE*);
        void ListAllEdges(void);
    private:
        EDGE_NODE *edgeHead;
        };

// edge.cpp
#include <iostream.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include "edge.hpp"


EdgeList::EdgeList()
        {
        edgeHead = (EDGE_NODE*)NULL;
//   cout << "EdgeList Constructor\n";
        }


EdgeList::~EdgeList()
        {
        while (edgeHead != (EDGE_NODE*)NULL)
                DelEdge(edgeHead);
//      cout << "EdgeList Destructor\n";
        }


EDGE_NODE* EdgeList::BuildNewEdge(int t, int w, int l, float dist)
        {
        EDGE_NODE *a;

//      if (otherVertex != (void*)NULL)
//              {
                a = (EDGE_NODE*)malloc(sizeof(EDGE_NODE));
                if (a == (EDGE_NODE*)NULL)
                        {
                        cout << "Out of Memory in BuildNewEdge()\n";
                        exit(0);
                        }
                (*a).t = t;
                (*a).w = w;
                (*a).l = l;
                (*a).dist = dist;
                (*a).prev = (*a).next = (EDGE_NODE*)NULL;
                return(a);
//              }
//      return((EDGE_NODE*)NULL);
        }


void EdgeList::DelEdge(EDGE_NODE*a)
   {
   EDGE_NODE*tmp;

//      cout << "edge 0\n";
        if (a != (EDGE_NODE*)NULL)
                {
                if ((a->prev == (EDGE_NODE*)NULL)  // del last remaining edge
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 1\n";
                        free(a);
                        edgeHead = (EDGE_NODE*)NULL;
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge in middle
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 2\n";
                        tmp = a->prev;
                        tmp->next = a->next;
                        tmp = a->next;
                        tmp->prev = a->prev;
                        free(a);
                        return;
                        }
                if ((a->prev == (EDGE_NODE*)NULL)  // del edge at sol
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 3\n";
```

```
                        tmp = a->next;
                        tmp->prev = (EDGE_NODE*)NULL;
                        edgeHead = tmp;
                        free(a);
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge at eol
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 4\n";
                        tmp = a->prev;
                        tmp->next = (EDGE_NODE*)NULL;
                        free(a);
                        return;
                        }
                }
        }


void EdgeList::DelEdgeToVertex(int t, int w, int l)
        {
        EDGE_NODE*tmp;

        for(tmp = GetFirstEdge(); tmp != (EDGE_NODE*)NULL; tmp = GetNextEdge(tmp))
                {
                if ((tmp->t == t)
        && (tmp->w == w)
        && (tmp->l == l))
                        {
                        DelEdge(tmp);
                        break;
                        }
                }
        }


EDGE_NODE* EdgeList::GetFirstEdge()
        {
   return(edgeHead);
   }


EDGE_NODE* EdgeList::GetNextEdge(EDGE_NODE*cur)
   {
   return(cur->next);
   }


void EdgeList::InsertNewEdge(EDGE_NODE *edge)
        {
        EDGE_NODE*cur;

        if (edge == (EDGE_NODE*)NULL)
                return;

        if (edgeHead == (EDGE_NODE*)NULL)
                {
                edgeHead = edge;
                edge->prev = edge->next = (EDGE_NODE*)NULL;
//              cout << "Inserted edge into empty list ";
//              cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//              cout << "Dist = " << edge->dist << "\n";
                return;
                }

        for(cur=edgeHead; cur != (EDGE_NODE*)NULL; cur=(*cur).next)
                {
                if ( (edge->t > cur->t)
                || ((edge->t == cur->t)
                        && (edge->w > cur->w))
                || ((edge->t == cur->t)
                        && (edge->w == cur->w)
                        && (edge->l > cur->l)) )
                        {                               // insert after cur
                        if (cur->next == (EDGE_NODE*)NULL)
                                {
                                edge->next = (EDGE_NODE*)NULL;// no more so append to eol
                                edge->prev = cur;
                                cur->next = edge;
                                break;
                                }
                        else
                                continue;                       // try next one
                        }
```

```
            if ( (edge->t < cur->t)
            || ((edge->t == cur->t)
                 && (edge->w < cur->w))
            || ((edge->t < cur->t)
                 && (edge->w == cur->w)
                 && (edge->l < cur->l)) )
                 {                                    // insert before cur
                 if (cur->prev == (EDGE_NODE*)NULL)
                      {
                      edge->prev = (EDGE_NODE*)NULL;  // at start of list
                      edge->next = cur;
                      cur->prev = edge;
                      edgeHead = edge;
                      break;
                      }
                 else
                      {
                      edge->prev = cur->prev;     // in middle/end of list
                      edge->next = cur;
                      cur->prev = edge;
                      cur = edge->prev;
                      cur->next = edge;
                      break;
                      }
                 }
            if ( (edge->t == cur->t)
            && (edge->w == cur->w)
            && (edge->l == cur->l) )
                 {
                 /* already here - replace it ! */
                 cout << "Already here - replacing values and disposing of new edge ";
        cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
        cout << "Dist = " << edge->dist << "\n";
                 cur->w = edge->w;
                 cur->l = edge->l;
                 cur->dist = edge->dist;
                 free(edge);
                 break;
                 }
            }
//      cout << "Inserted edge into list ";
//      cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//      cout << "Dist = " << edge->dist << "\n";
        }


void EdgeList::ListAllEdges()
        {
        EDGE_NODE *edge;

        for(edge = edgeHead; edge != (EDGE_NODE*)NULL; edge = (*edge).next)
             {
             cout << "Edge to (";
             cout << (*edge).t << ",";
             cout << (*edge).w << ",";
             cout << (*edge).l << "). Dist is " << (*edge).dist << "\n";
             }
        cout << "-----------------\n";
        }

// object.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define NUM_DIRS 7 /* this is the 8 horizontal directions: (0->7) */
enum directions { front, back, left, right, frontright, frontleft, backright, backleft, up, down, clear};


typedef struct o
        {
        int direction, velocity;
        int w, l;
        struct o *prev, *next;
        } OBJECT_NODE;
```

```
class ObjectList
        {
        public:
                ObjectList();
                ~ObjectList();
                OBJECT_NODE* BuildNewObject(int, int);
        void DelAllObjects(void);
        void DelObject(OBJECT_NODE*);
        OBJECT_NODE* GetNextObject(void);
                void InsertNewObject(OBJECT_NODE*);
        void ListAllObjects(void);
        private:
                OBJECT_NODE *objectHead;
        };

// object.cpp
#include <iostream.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
// #include "edge.hpp"
#include "object.hpp"

ObjectList::ObjectList()
        {
        objectHead = (OBJECT_NODE*)NULL;
        cout << "ObjectList Constructor\n";
        }


ObjectList::~ObjectList()
        {
        cout << "ObjectList Destructor\n";
        DelAllObjects();
        }


OBJECT_NODE* ObjectList::BuildNewObject(int w, int l)
        {
        OBJECT_NODE *newPtr;

        newPtr = (OBJECT_NODE*)malloc(sizeof(OBJECT_NODE));
        if (newPtr == (OBJECT_NODE*)NULL)
                {
                cout << "Out of memory in BuildNewObject()\n";
                return(NULL);
                }
        (*newPtr).direction = rand() % NUM_DIRS;
        (*newPtr).velocity = 1;
        (*newPtr).prev = (*newPtr).next = (OBJECT_NODE*)NULL;
        (*newPtr).w = w;
        (*newPtr).l = l;
        return(newPtr);
        }


void ObjectList::DelAllObjects()
        {
        OBJECT_NODE *tmp;

        while (objectHead != (OBJECT_NODE*)NULL)
                {
                tmp = objectHead;
                objectHead = (*objectHead).next;
                DelObject(tmp);
                }
        }


void ObjectList::DelObject(OBJECT_NODE *todie)
   {
        OBJECT_NODE *cur;

        if (((*todie).prev != (OBJECT_NODE*)NULL)
        && ((*todie).next != (OBJECT_NODE*)NULL))
                {
//              cout << "Deleted Object (" << todie->w << "," << todie->l << ")\n";
                cur=(*todie).prev;
                if (cur == todie)
```

```
                        {
                        objectHead = (OBJECT_NODE*)NULL;
//                      cout << "Object List Empty\n";
                        free(todie);
                        return;
                        }
                else
                        {
                        (*cur).next=(*todie).next;
                        cur = (*todie).next;
                        (*cur).prev = (*todie).prev;
                        if (objectHead == todie)
                                objectHead = (*todie).next;
        free(todie);
          return;
                        }
                }
        cout << "***Did NOT delete rotton Object (" << todie->w << "," << todie->l << ")\n";
        }


OBJECT_NODE* ObjectList::GetNextObject()
    {
    OBJECT_NODE*tmp;

        if (objectHead == (OBJECT_NODE*)NULL)
                return((OBJECT_NODE*)NULL);

        tmp = objectHead;
    objectHead = objectHead->next;
    return(tmp);
    }


void ObjectList::InsertNewObject(OBJECT_NODE*newPtr)
        {
        OBJECT_NODE *cur;

        if (newPtr == (OBJECT_NODE*)NULL)
                return;

        if (objectHead == (OBJECT_NODE*)NULL)
                {
                objectHead = newPtr;
                (*newPtr).prev = (*newPtr).next = newPtr;
                }
        else
                {
                /* insert before first node */
                cur = objectHead;
                (*newPtr).next = cur;
                (*newPtr).prev = (*cur).prev;
                (*cur).prev = newPtr;

                cur = (*newPtr).prev;
                (*cur).next = newPtr;
                objectHead = newPtr;
        }
//      cout << "Inserted object (" << newPtr->w << "," << newPtr->l << ")\n";
        }


void ObjectList::ListAllObjects()
        {
   OBJECT_NODE *tmp;

   if (objectHead == (OBJECT_NODE*)NULL)
       {
       cout << "ObjectList is empty\n";
       return;
       }
   for(tmp=objectHead;;tmp = tmp->next)
       {
        cout << "Object:(" << tmp->w << "," << tmp->l << ")\n";
       tmp = tmp->next;
       if (tmp == objectHead)
          {
          cout << "ObjectList ended\n";
          break;
          }
       }
        }
```

```
// vertex.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00


typedef struct vn
        {
        int t,w,l;
        char nodeType;
        struct vn *searchPrev, *searchNext;
        float searchDist;
        int searchMarker;
        EdgeList *edgeList;
        struct vn *pathFrom, *pathTo;
        struct vn *prev, *next;
        } VERTEX_NODE;


class VertexList
        {
        public:
                VertexList();
                ~VertexList();
                void AddToSearchList(VERTEX_NODE*);
        VERTEX_NODE* BuildNewVertex(int,int,int,char);
                int CalcRobotDir(VERTEX_NODE*);
                void DelAllVertices(void);
        void DelVertex(VERTEX_NODE*);
        int FindPath(void);
        VERTEX_NODE* FindVertex(int, int, int);
                VERTEX_NODE* GetFirstVertex(void);
                VERTEX_NODE* GetNextVertex(VERTEX_NODE*);
        VERTEX_NODE* GetStartVertex(void);
        void InsertAllVertices(void);
                void InsertNewVertex(VERTEX_NODE*);
        void ListAllVertices(void);
        void ListSearchList(void);
//              void MoveRobot(int, int, int, int);
                void MarkPath(VERTEX_NODE*);
                void RemoveFromSearchList(VERTEX_NODE*);
                void TrimSearchList(void);
        private:
                VERTEX_NODE* vertexHead;
        VERTEX_NODE* searchHead;
        int searchMarker;
        float searchTrimDist;
        };

// vertex.cpp
#include <iostream.h>
#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include "edge.hpp"
#include "vertex.hpp"

VertexList::VertexList()
        {
        vertexHead = (VERTEX_NODE*)NULL;
    searchHead = (VERTEX_NODE*)NULL;
    searchMarker = 0;
    searchTrimDist = -1.0;
    cout << "Initialised Vertex class\n";
        }


VertexList::~VertexList()
        {
```

```
        cout << "VertexList destructor started\n";
        DelAllVertices();
        cout << "VertexList destructor ended!\n";
        }


void VertexList::AddToSearchList(VERTEX_NODE*a)
        {
        VERTEX_NODE *tmp;

        if (a == (VERTEX_NODE*)NULL)
                return;

        if (((*a).searchPrev != (VERTEX_NODE*)NULL) // if already in fringe
        || ((*a).searchNext != (VERTEX_NODE*)NULL)) // list, dont add again
                return;

        if (searchHead == (VERTEX_NODE*)NULL)
                {
                searchHead = a;
                a->pathFrom = a->pathTo = (VERTEX_NODE*)NULL;
                a->searchPrev = a->searchNext = a;

//              cout << "Inserted into empty search list (";
//              cout << a->t << "," << a->w << "," << a->l << ")\n";
                }
        else
                {
                a->searchNext = searchHead;
                a->searchPrev = searchHead->searchPrev;
                searchHead->searchPrev = a;
                tmp = a->searchPrev;
                tmp->searchNext = a;
//              cout << "Appended to search list (";
//              cout << a->t << "," << a->w << "," << a->l << ")\n";
                }
        }


VERTEX_NODE* VertexList::BuildNewVertex(int t, int w, int l, char nodeType)
        {
        VERTEX_NODE *newPtr;

        newPtr = (VERTEX_NODE*)malloc(sizeof(VERTEX_NODE));
        if (newPtr == (VERTEX_NODE*)NULL)
                {
                cout << "Malloc() failed in BuildNewVertex!\n";
                exit(0);
                }
        (*newPtr).t = t;
        (*newPtr).w = w;
        (*newPtr).l = l;
        (*newPtr).nodeType = nodeType;
        (*newPtr).edgeList = new EdgeList();
        (*newPtr).pathTo = (*newPtr).pathFrom = (VERTEX_NODE*)NULL;
        (*newPtr).searchDist = 0.0;
        (*newPtr).searchPrev = (*newPtr).searchNext = (VERTEX_NODE*)NULL;
        (*newPtr).prev = (*newPtr).next = (VERTEX_NODE*)NULL;
        if (nodeType == START)
                {
                (*newPtr).searchMarker = searchMarker+1;
                AddToSearchList(newPtr);
                }
        else
                (*newPtr).searchMarker = searchMarker;
        return(newPtr);
        }


void VertexList::DelAllVertices()
   {
        while (vertexHead != (VERTEX_NODE*)NULL)
                {
//              cout << "About to delete (" << vertexHead->t << "," << vertexHead->w ;
//              cout << "," << vertexHead->l << ") at addr: " << vertexHead ;
//              cout << ": Prev:" << vertexHead->prev << ": Next:" << vertexHead->next <<"\n";
                DelVertex(vertexHead);
                }
        cout << "----------\n";
//      getch();
        }
```

```
void VertexList::DelVertex(VERTEX_NODE *todie)
        {
        VERTEX_NODE *tmp;
        EDGE_NODE*a;

        if (todie == (VERTEX_NODE*)NULL)
                return;

        for(a = todie->edgeList->GetFirstEdge();
                a != (EDGE_NODE*)NULL;
                a = todie->edgeList->GetFirstEdge())
                {
       todie->edgeList->DelEdgeToVertex(a->t, a->w, a->l);
       tmp = FindVertex(a->t, a->w, a->l);
       if (tmp != (VERTEX_NODE*)NULL)
          {
          tmp->edgeList->DelEdgeToVertex(todie->t, todie->w, todie->l);
          }
//              tmp = (VERTEX_NODE*)(a->otherVertex);
//              cout << "On vertex (" << todie->t << "," << todie->w << ",";
//              cout << todie->l << "):Del edge to (" << tmp->t << "," << tmp->w << ",";
//              cout << tmp->l << ")\n";
//              cout << "todie at:" << todie << ". a at:" << a << "\n";
//              cout << "  tmp at:" << tmp << "\n";
//              todie->edgeList->DelEdgeToVertex((void*)tmp);
//              cout << "On vertex (" << tmp->t << "," << tmp->w << ",";
//              cout << tmp->l << "):Del edge to (" << todie->t << "," << todie->w << ",";
//              cout << todie->l << ")\n";
//              tmp->edgeList->DelEdgeToVertex((void*)todie);
                }
        delete todie->edgeList;
        RemoveFromSearchList(todie);

        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
                {
                vertexHead = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
                tmp = (*todie).prev;
                (*tmp).next = (*todie).next;
                tmp = (*todie).next;
                (*tmp).prev = (*todie).prev;
                free(todie);
                return;
                }
        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
                vertexHead = tmp = (*todie).next;
                (*tmp).prev = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
                {
                tmp = (*todie).prev;
                (*tmp).next = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        }


int VertexList::FindPath()
   {
        EDGE_NODE *e;
        int goalFound=FALSE;
        float dist;
        VERTEX_NODE *cur, *adj;

        if (searchHead->nodeType == START)
                searchMarker = searchHead->searchMarker;
        else
                return(FALSE);
        for (cur = searchHead; cur != (VERTEX_NODE*)NULL; cur = searchHead)
                {
                if ((cur->searchDist >= searchTrimDist)
                && (searchTrimDist > 0.0))
```

```
			{
			RemoveFromSearchList(cur);
			continue;
			}
		for(e = cur->edgeList->GetFirstEdge();
			e != (EDGE_NODE*)NULL;
			e = cur->edgeList->GetNextEdge(e))
			{
	adj = FindVertex(e->t, e->w, e->l);
	if (adj == (VERTEX_NODE*)NULL)
	   continue;

//	  adj = (VERTEX_NODE*) e->otherVertex;
			dist = cur->searchDist + e->dist;
			if ( (adj->searchMarker != searchMarker)
			|| ((adj->searchMarker == searchMarker)
				&& (adj->searchDist > dist)) )
				{
				if ((dist < searchTrimDist)
				|| (searchTrimDist <= 0.0))
					{
					adj->pathFrom = cur;
					adj->searchMarker = searchMarker;
					adj->searchDist = dist;
					AddToSearchList(adj);
					if (adj->nodeType == GOAL)
						{
						goalFound = TRUE;
						searchTrimDist = dist;
						MarkPath(adj);
						TrimSearchList();
						}
					}
				}
			}
		RemoveFromSearchList(cur);
		}
	return(goalFound);
	}


VERTEX_NODE* VertexList::FindVertex(int t, int w, int l)
   {
   VERTEX_NODE* cur;

   for(cur = GetFirstVertex();
      cur != (VERTEX_NODE*)NULL;
      cur = GetNextVertex(cur))
      {
      if ((cur->t == t)
      && (cur->w == w)
      && (cur->l == l))
         return(cur);		// found it

      if ((cur->t >= t)
      && (cur->w >= w)
      && (cur->l >= l))
         break;			// passed it - it's not in the list
      }
   return((VERTEX_NODE*)NULL);
   }


VERTEX_NODE* VertexList::GetFirstVertex()
   {
   return(vertexHead);
   }


VERTEX_NODE* VertexList::GetNextVertex(VERTEX_NODE*cur)
   {
   return(cur->next);
   }


VERTEX_NODE* VertexList::GetStartVertex()
   {
   VERTEX_NODE*tmp;

   for(tmp = GetFirstVertex();
      tmp != (VERTEX_NODE*)NULL;
      tmp = GetNextVertex(tmp))
      {
```

```
            if (tmp->nodeType == START)
        break;
        }
    return(tmp);
    }


void VertexList::InsertNewVertex(VERTEX_NODE*a)
        {
        VERTEX_NODE *cur;

        if (a == (VERTEX_NODE*)NULL)
                return;

        if (vertexHead == (VERTEX_NODE*)NULL)
                {
                vertexHead = a;
                (*a).prev = (*a).next = (VERTEX_NODE*)NULL;
                return;
                }

        for(cur=vertexHead; cur != (VERTEX_NODE*)NULL;   cur=(*cur).next)
                {
                if ( (a->t > cur->t)
        || ((a->t == cur->t)
           && (a->w > cur->w))
         || ((a->t == cur->t)
           && (a->w == cur->w)
                && (a->l > cur->l)) )
                        {                               // insert after cur
                        if ((*cur).next == (VERTEX_NODE*)NULL)
                                {
                                (*a).next = (*cur).next;    // eol - append new node
                                (*a).prev = cur;
                                (*cur).next = a;
                                break;
                                }
                        else
                                continue;                   // get next node
                        }
                if ( (a->t < cur->t)
        || ((a->t == cur->t)
           && (a->w < cur->w))
        || ((a->t == cur->t)
                    && (a->w == cur->w)
                        && (a->l < cur->l)) )
                        {                               // insert before cur
                        if (cur->prev == (VERTEX_NODE*)NULL)
                                {
                                a->prev = (VERTEX_NODE*)NULL;// at start of list
                                a->next = cur;
                                cur->prev = a;
                                vertexHead = a;
                                break;
                                }
                        else
                                {
                                a->prev = cur->prev;        // in middle/end of list
                                a->next = cur;
                                cur->prev = a;
                                cur = a->prev;
                                cur->next = a;
                                break;
                                }
                        }
                if ((a->t == cur->t)
        && (a->w == cur->w)
                && (a->l == cur->l))
                        {                               // insert after cur at eol
                        if ((*cur).next == (VERTEX_NODE*)NULL)
                                {
                                (*a).next = (VERTEX_NODE*)NULL;
                                (*a).prev = cur;
                                (*cur).next = a;
                                break;
                                }
                        else
                                continue;
                        }
                }
//      cout << "Inserted vertex (";
// cout << (*a).t << "," << (*a).w << "," << (*a).l << ")\n";
        }
```

```
void VertexList::ListAllVertices()
	{
	VERTEX_NODE *cur=vertexHead;

    while (cur != (VERTEX_NODE*)NULL)
	{
		cout << "Vertex:" <<  (*cur).nodeType << ": at (";
		cout << (*cur).t << "," << (*cur).w << "," << (*cur).l << ")\n";
		cur->edgeList->ListAllEdges();
	cur = (*cur).next;
	}
    }


void VertexList::ListSearchList()
	{
	VERTEX_NODE*cur;

	if (searchHead == (VERTEX_NODE*)NULL)
		{
		cout << "Empty SearchList! (marker=" << searchMarker << ")\n";
		return;
		}

	cout << "List of nodes in Search list (marker=" << searchMarker << ")\n";
	cur = searchHead;
	do
		{
		cout << "SearchList node (" << cur->t << "," << cur->w << "," << cur->l << ") ";
      cout << "Dist = " << cur->searchDist << "\n";
		cur=cur->searchNext;
		}
	while (cur != searchHead);
	cout << "End of SearchList\n";
	}


void VertexList::MarkPath(VERTEX_NODE*v)
	{
	VERTEX_NODE*tmp;

	if (v == (VERTEX_NODE*)NULL)
		return;

//	cout << "Path from Goal to Start\n";
	v->pathTo = (VERTEX_NODE*)NULL;
	do
		{
		tmp = v->pathFrom;
//		cout << "(" << v->t << "," << v->w << "," << v->l << ")\n";
		if (tmp != (VERTEX_NODE*)NULL)
			{
			tmp->pathTo = v;
			v = v->pathFrom;
			}
		}
	while (tmp != (VERTEX_NODE*)NULL);
	}


void VertexList::RemoveFromSearchList(VERTEX_NODE*a)
	{
	VERTEX_NODE *cur;

	if ((a->searchPrev == (VERTEX_NODE*)NULL)
	|| (a->searchNext == (VERTEX_NODE*)NULL))
	{
	if (searchHead == a)
	   searchHead = (VERTEX_NODE*)NULL;
	return;
	}

	cur=a->searchPrev;
	if (cur == a)
		{
	searchHead = (VERTEX_NODE*)NULL;
//		cout << "Removed Search Node (";
//	cout << a->t << "," << a->w << "," << a->l << "). SearchList empty\n";
		}
	else
		{
		cur->searchNext = a->searchNext;
		cur = a->searchNext;
```

```
                cur->searchPrev = a->searchPrev;
                if (searchHead == a)
                        searchHead = a->searchNext;
//              cout << "Removed Search Node (";
//      cout << a->t << "," << a->w << "," << a->l << ").\n";
                }
        a->searchPrev = a->searchNext = (VERTEX_NODE*)NULL;
        }


void VertexList::TrimSearchList()
    {
        VERTEX_NODE *cur, *tmpPtr=(VERTEX_NODE*)0;
        int trimmed=TRUE;

//      cout << "Trimming search list to " << searchTrimDist << " or less\n";
        if ((searchHead == (VERTEX_NODE*)NULL)
    || (searchTrimDist < 0.0))
                return;

    cur = searchHead;
        while ((tmpPtr != cur) || (trimmed == TRUE))
                {
                if (trimmed==TRUE)
                        {
                        tmpPtr=cur;
                        trimmed=FALSE;
                        }
                if ((*cur).searchDist >= searchTrimDist)
                        {
//                      cout << "Trimmed out (";
//              cout << cur->t << "," << cur->w << "," << cur->l << ") ";
//              cout << "Dist was " << cur->searchDist << "\n";
                        if (cur == cur->searchNext)
                                {
             RemoveFromSearchList(cur);
                                break;
                                }
                        else
                                {
                                tmpPtr = cur->searchNext;
                                RemoveFromSearchList(cur);
                                cur = tmpPtr;
                                trimmed = TRUE;
                                }
                        }
                else
                        cur = cur->searchNext;
                }
//    cout << "-------\n";
    }
```

B.4. Standard Depth First Graph Theory

The Standard Depth First Search Algorithm for Graph Theory which was implemented as part of this project was coded in ANSI C++. The filenames for each of the separate source code files were supplied inside C++ format comments (i.e. //) at the beginning of each file listing. A detailed explanation of the design behind this program was presented in Chapter 4.

```
// main.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>            // for getch()
#include <alloc.h>            // for coreleft()
#include <stdlib.h>           // for itoa()
#include <string.h>           // for strcpy()
#include <sys\timeb.h>
#include <dir.h>
#include <ctype.h>
#include "edge.hpp"
#include "vertex.hpp"
#include "object.hpp"
#include "domain.hpp"
#include "bench.hpp"
```

```
#define MAPFILE_MASK "MAP*.DAT"

int main(int, char**);
void BuildVertexListFromDomain(VertexList&, Domain&);
int CalcRobotDir(VERTEX_NODE*);
void FindAllEdges(VertexList&, Domain&);
void MemStatus(char*);
int SetWorkingDir(void);
void RenameMapFile(char*);


int main(int argc, char** argv)
	{
	char* shortName;
	struct ffblk mapfile;

	for(;;)
		{
		shortName = strstr(argv[0], "\\");
		if (shortName == (char*)NULL)
			{
			shortName = argv[0];
			cout << "This is " << shortName << "\n";
			break;
			}
		else
			argv[0] = shortName+1;
		}

	MemStatus("Free memory before mainloop in main: ");

	if (!SetWorkingDir())
		{
		cout << "Program exiting gracefully\n";
		return(1);
		}
	if (findfirst(MAPFILE_MASK, &mapfile, 0))
		{
		cout << "Program exiting gracefully - no map files found\n";
		return(1);
		}
	do
		{
		char tmp[256];
		int i;
		Benchmark stopWatch(shortName);

		sprintf(tmp, "Starting on %s: ", mapfile.ff_name);
		MemStatus(tmp);
		for (i=0; i<NUMTIMES; i++)
			{
			int flag, timeTaken;
			float distTravelled;
			Domain theWorld(10, 20, 20, mapfile.ff_name);

			timeTaken = 0;
			distTravelled = 0.0;
			stopWatch.IterStart(i, mapfile.ff_name);
			stopWatch.Click();
			for(;;)
				{
				VertexList vertList;

				BuildVertexListFromDomain(vertList, theWorld);
				FindAllEdges(vertList, theWorld);
				stopWatch.Click();
				theWorld.DrawDomain();
				stopWatch.Click();
				flag = vertList.FindPath();
				stopWatch.Click();
				if (flag == TRUE)
					cout << "Path found the GOAL!\n";
				else
					cout << "No path found to GOAL!\n";

				if (flag)
					{
					VERTEX_NODE*t1, *t2;

					t1 = vertList.GetStartVertex();
					if (t1 != (VERTEX_NODE*)NULL)
						{
						t2 = t1->pathTo;
						if (t2 != (VERTEX_NODE*)NULL)
```

```
                                                        {
                                                        if ((t1->w == t2->w)
                                                        && (t1->l == t2->l)
                                                        && (t2->nodeType == GOAL))
                                                              {
                                                              cout << "MADE IT TO THE GOAL!\n";
                                                              break;
                                                              }
                                                        distTravelled += theWorld.MoveRobot(t1->t, t1->w, t1->l,

                              t2->t, t2->w, t2->l);
                                                        }
                                                  }
                                            }
                                    theWorld.AdvanceTime();
                                    timeTaken++;
                                    }
                              stopWatch.IterStop(timeTaken, distTravelled);
                              }
                        stopWatch.LogCalcs();
                        RenameMapFile(mapfile.ff_name);
//                      break;
                        }
            while (!findnext(&mapfile));

            MemStatus("Free memory after mainloop in main: ");
            return(0);
            }


void BuildVertexListFromDomain(VertexList &vList, Domain &domain)
            {
            char type;
            int t=domain.GetDomainTimeSlices(), a;
            int w=domain.GetDomainWidth(), b;
            int l=domain.GetDomainLength(), c;
            VERTEX_NODE*newPtr;

            for(a=0; a<t; a++)
              {
                  for(b=0; b<w; b++)
                        {
                        for(c=0; c<l; c++)
                              {
                              type = domain.GetPointType(a, b, c);
                              if ((type == VERTEX)
                              || (type == START)
                              || (type == GOAL))
                                    {
                                    newPtr = vList.BuildNewVertex(a, b, c, type);
                                    vList.InsertNewVertex(newPtr);
                                    }
                              }
                        }
                  }
            }


void FindAllEdges(VertexList &vList, Domain&domain)
            {
            EDGE_NODE*e;
            VERTEX_NODE*a,*b;
            float dist;

            for(a = vList.GetFirstVertex();
                  a != (VERTEX_NODE*)NULL;
                  a = vList.GetNextVertex(a))
                  {
                  for(b = vList.GetNextVertex(a);
                        b != (VERTEX_NODE*)NULL;
                        b = vList.GetNextVertex(b))
              {
              if (a == b)
                              continue;

                        if ((a->t == b->t)
                        && (a->w == b->w)
                        && (a->l == b->l))
                              continue;

              dist = domain.CheckLine(a->t, a->w, a->l, b->t, b->w, b->l);
              if (dist > 0.0)
                 {
                              e = a->edgeList->BuildNewEdge(b->t, b->w, b->l, dist);
```

```
                                a->edgeList->InsertNewEdge(e);
                                }
                        dist = domain.CheckLine(b->t, b->w, b->l, a->t, a->w, a->l);
                        if (dist > 0.0)
                                {
                                e = b->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                                b->edgeList->InsertNewEdge(e);
                }
            }
                }
        }


void MemStatus(char *StatusMessage)
        {
        char tmp[256];
        fstream debugFile;
        long MemLeft;

        debugFile.open("DEBUG.LOG", ios::app);
        if (!debugFile)
                cout << "Unable to open DEBUG.LOG\n";

        MemLeft = (long) coreleft();
        sprintf(tmp, "%s%ld\n", StatusMessage, MemLeft);
        debugFile.write(tmp, strlen(tmp));
        debugFile.close();
//      cout << StatusMessage << MemLeft << "\n";
        }


int SetWorkingDir()
        {
        char mapdir[256];

        cout <<"Enter the directory containing map files, or \"q\" for quit:";
        cin >> mapdir;
        if (toupper(mapdir[0]) == 'Q')
                {
                cout << "Quitting...\n";
                return(FALSE);
                }
        if (chdir(mapdir))
                {
                cout << "The directory " << mapdir << " could not be found.\n";
                return(FALSE);
                }
        cout << "Made " << mapdir << " the current directory.\n";
        return(TRUE);
        }


void RenameMapFile(char*filename)
        {
        char newfilename[128];
        char* ch;
        int i=(int)'.';

        strcpy(newfilename, filename);
        ch = strrchr(newfilename,i);
        if (ch != (char*)NULL)
                {
                strcpy(ch, ".bak");
                rename(filename, newfilename);
                }
        else
                exit(1);
        }

// bench.hpp
#define FALSE 0
#define TRUE !FALSE
#define NUMTIMES 10
#define MAXFILENAME 13

class Benchmark
        {
        public:
                Benchmark(char*);
                ~Benchmark();
                void Click(void);
                void IterStart(int, char*);
                void IterStop(int, float);
                void LogCalcs(void);
```

```
	private:
		void Diff(struct timeb*, struct timeb*, struct timeb*);

		fstream logFile;
		fstream avgFile;
		char mapFileName[MAXFILENAME];
		struct timeb benchmarks[NUMTIMES][2];
		float distRobotTravelled[NUMTIMES];
		int timeTaken[NUMTIMES];
		int currentIter;
		int clickToggleFlag;
		struct timeb elapsedTime, computeTime;
		struct timeb startTime, clickOnTime, clickOffTime;
	};

// bench.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <alloc.h>		// for coreleft()
#include <stdlib.h>		// for itoa()
#include <string.h>		// for strcpy()
#include <sys\timeb.h>
#include "bench.hpp"


Benchmark::Benchmark(char *fileName)
	{
	char *tmp;
	char logname[MAXFILENAME];

	while((tmp = strchr(fileName, '\\')) != (char*)NULL)
		fileName = tmp+1;
	strcpy(logname, fileName);
	if ((tmp = strchr(logname, '.')) != (char*)NULL)
		strcpy(tmp, ".LOG");
	else
		strcat(logname, ".LOG");
	logFile.open(logname, ios::app);
	if (!logFile)
		cout << "Unable to open " << logname << "\n";

	if ((tmp = strchr(logname, '.')) != (char*)NULL)
		strcpy(tmp, ".AVG");
	else
		strcat(logname, ".AVG");
	avgFile.open(logname, ios::app);
	if (!avgFile)
		cout << "Unable to open " << logname << "\n";

	cout << "Initialised Benchmark class\n";
	}


Benchmark::~Benchmark()
	{
	if (logFile)
		{
		logFile.flush();
		logFile.close();
		}
	if (avgFile)
		{
		avgFile.flush();
		avgFile.close();
		}
	cout << "Closed log files and Destroying Benchmark class\n";
	}


void Benchmark::Click()
	{
	switch(clickToggleFlag)
		{
		case TRUE:
			ftime(&clickOnTime);
			clickToggleFlag = FALSE;
			break;
		case FALSE:
		default:
			ftime(&clickOffTime);
			Diff(&clickOnTime, &clickOffTime, &computeTime);
			clickToggleFlag = TRUE;
		}
```

```
        }

void Benchmark::Diff(struct timeb*start, struct timeb*stop, struct timeb*diff)
        {
        if ((*stop).millitm < (*start).millitm)
                {
                (*stop).millitm += (short)1000; /* carry when subtracting, stops*/
                (*start).time += 1L;            /* negative wraparound problems!*/
                }
        (*diff).millitm += (*stop).millitm - (*start).millitm;
        (*diff).time += (long)((*diff).millitm / (short)1000);
        (*diff).millitm %= (short)1000;
        (*diff).time += ((*stop).time-(*start).time);
        }


void Benchmark::IterStart(int i, char*s)
        {
        currentIter = i;
        clickToggleFlag = TRUE;
        computeTime.time = elapsedTime.time = 0L;
        computeTime.millitm = elapsedTime.millitm = 0;

        strcpy(mapFileName, s);

        ftime(&startTime);
        }


void Benchmark::IterStop(int t, float distTravelled)
        {
        struct timeb stopTime;

        ftime(&stopTime);
        Diff(&startTime, &stopTime, &elapsedTime);

        benchmarks[currentIter][0].time = elapsedTime.time;
        benchmarks[currentIter][0].millitm = elapsedTime.millitm;
        benchmarks[currentIter][1].time = computeTime.time;
        benchmarks[currentIter][1].millitm = computeTime.millitm;
        timeTaken[currentIter] = t;
        distRobotTravelled[currentIter] = distTravelled;
        }



void Benchmark::LogCalcs()
        {
        char tmp[256];
        float avgDist;
        struct timeb avg;
        int i, avgTimeTaken;

        for(i=0, avg.time=0L, avg.millitm=0, avgDist=0.0, avgTimeTaken=0;
                i<NUMTIMES;
                i++)
                {
                sprintf(tmp, "%s (%02d) Elapsed time:%05ld.%03d\n", mapFileName, i,
                                benchmarks[i][0].time, benchmarks[i][0].millitm);
                logFile.write(tmp, strlen(tmp));
                sprintf(tmp, "%s (%02d) Compute time:%05ld.%03d\n", mapFileName, i,
                                benchmarks[i][1].time, benchmarks[i][1].millitm);
                logFile.write(tmp, strlen(tmp));
                sprintf(tmp, "%s (%02d) Dist travelled:%f\n", mapFileName, i,
                                distRobotTravelled[i]);
                logFile.write(tmp, strlen(tmp));
                sprintf(tmp, "%s (%02d) Time Slices Taken:%d\n", mapFileName, i,
                                timeTaken[i]);
                logFile.write(tmp, strlen(tmp));
                avg.time+=benchmarks[i][0].time;
                avg.millitm+=benchmarks[i][0].millitm;
                if (avg.millitm % 1000 != avg.millitm)
                        {
                        avg.time += (long)(avg.millitm / (short)1000);
                        avg.millitm %= (short)1000;
                        }
                }

       sprintf(tmp, "%s Tot. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
       logFile.write(tmp, strlen(tmp));
       cout << avg.time << "." << avg.millitm << "\n";
       cout << tmp << "\n";
```

```
        i = (int) (avg.time % (long)NUMTIMES);
        avg.time /= (long)NUMTIMES;
        avg.millitm += i * 1000;
        avg.millitm /= NUMTIMES;

        sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));

        for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
                {
                avg.time+=benchmarks[i][1].time;
                avg.millitm+=benchmarks[i][1].millitm;
                if (avg.millitm % 1000 != avg.millitm)
                        {
                        avg.time += (long)(avg.millitm / (short)1000);
                        avg.millitm %= (short)1000;
                        }
                }
        sprintf(tmp, "%s Tot. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));
        i = (int) (avg.time % (long)NUMTIMES);
        avg.time /= (long)NUMTIMES;
        avg.millitm += i * 1000;
        avg.millitm /= NUMTIMES;
        sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));

        for(i=0, avgTimeTaken=0; i<NUMTIMES; i++)
                avgTimeTaken+=timeTaken[i];
        avgTimeTaken = avgTimeTaken / NUMTIMES;
        sprintf(tmp, "%s Avg. Time Taken:%d\n", mapFileName, avgTimeTaken);
        avgFile.write(tmp, strlen(tmp));
        logFile.write(tmp, strlen(tmp));

        for(i=0, avgDist=0.0; i<NUMTIMES; i++)
                avgDist += distRobotTravelled[i];
        avgDist = avgDist / ((float)NUMTIMES);
        sprintf(tmp, "%s Avg. Dist Travelled:%f\n", mapFileName, avgDist);
        avgFile.write(tmp, strlen(tmp));
        logFile.write(tmp, strlen(tmp));

        sprintf(tmp, "%s =====================\n", mapFileName);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s =====================\n", mapFileName);
        logFile.write(tmp, strlen(tmp));
        }

// domain.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00

#define FROM_FRONT 0
#define FROM_BACK 1
#define FROM_LEFT 2
#define FROM_RIGHT 3
#define FROM_FRONTRIGHT 4
#define FROM_FRONTLEFT 5
#define FROM_BACKRIGHT 6
#define FROM_BACKLEFT 7
#define FROM_UP 8
#define FROM_DOWN 9
#define FROM_NOWHERE 10

#define NOCOST (float)0.0
#define NORMAL (float)1.0
```

```
#define NORMAL_DIAG (float)1.414214
#define BLOCKED (float)1000.0

typedef struct
        {
        float dist;
        int from;
        char type;
        float cost[9];
        } POINT;

class Domain
        {
        public:
                Domain(int, int, int, char*);
                ~Domain();
                void AdvanceTime(void);
        float CheckLine(int, int, int, int, int, int);
                void DrawDomain(void);
        int GetDomainLength(void) { return(domainLength); }
        int GetDomainTimeSlices(void) { return(domainTimeSlices); }
        int GetDomainWidth(void) { return(domainWidth); }
                POINT* GetPoint(int, int, int);
        float GetPointCost(int, int, int, int);
                char GetPointFrom(int, int, int);
                char GetPointType(int, int, int);
                int IsPointClear(int, int, int);
                int IsPointGoal(int, int, int);
                int IsPointNearObject(int, int, int);
                int IsPointObject(int, int, int);
                int IsPointStart(int, int, int);
        int IsPointVertex(int, int, int);
                float MoveRobot(int, int, int, int, int, int);
                void SetPointFrom(int, int, int, int);
                void SetPointType(int, int, int, char);
        private:
                void AgeTimeSlices(void);
        int CalcRobotDir(int, int, int, int, int, int);
                int ClearAdjPointOK(int, int, int);
                void ClearMobileObject(int, int, int);
                void ClearVerticesInTimeSlice(int);
                void DrawTimeSlice(int);
                void InitTimeSlice(int);
                void MarkMobileObject(int, int, int);
                void MoveMobileObject(OBJECT_NODE*);
                void MoveMobileObjects(void);
                void SetAdjObjsInTimeSlice(int);
                void SetGoalFromFile(char*);
                void SetMobileObjsFromFile(char*);
                void SetPermObjsInTimeSlice(int);
                void SetStartFromFile(char*);
                void SetVerticesInTimeSlice(int);

                ObjectList objList;
                POINT** domainHead;
        int domainWidth;
        int domainLength;
        int domainTimeSlices;
        };

// domain.cpp
#include <iostream.h>
#include <fstream.h>
#include <alloc.h>          // for coreleft()
#include <stdio.h>
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <conio.h>
#include "object.hpp"
#include "domain.hpp"

Domain::Domain(int numTimeSlices, int width, int length, char*mapFileName)
        {
        int i;

        cout << "Contructing Domain class\n";
        domainTimeSlices = numTimeSlices;
        domainWidth = width;
        domainLength = length;

        cout << "FreeHeap:" << farcoreleft() << "\n";
        cout << "Amount needed for one timeslice:";
        cout << width*length*sizeof(POINT) << "\n";
```

```
	domainHead = (POINT**)farcalloc(numTimeSlices, sizeof(POINT**));
	if (domainHead == (POINT**)NULL)
		{
		cout << "Not enough memory to build model of world\n";
		domainHead = (POINT***)NULL;
		domainTimeSlices = domainWidth = domainLength = 0;
		return;
		}

	for(i=0; i<numTimeSlices; i++)
		{
		domainHead[i] = (POINT*)farcalloc(width*length, sizeof(POINT));
		cout << "FreeHeap after timeslice allocated:" << farcoreleft() << "\n";
		if (domainHead[i] == (POINT*)NULL)
			{
			if (i==0)
				cout << "Not enough memory to build model of world\n";
			else
				cout << "Only enough memory to build " << i << " of the ";
				cout << numTimeSlices << " timeslices in model of world\n";
			domainTimeSlices = i;
			break;
			}
		InitTimeSlice(i);
		SetPermObjsInTimeSlice(i);
		}
	SetStartFromFile(mapFileName);
	SetGoalFromFile(mapFileName);
	SetMobileObjsFromFile(mapFileName);
	for(i=0; i<domainTimeSlices; i++)
		{
		SetAdjObjsInTimeSlice(i);
		SetVerticesInTimeSlice(i);
		}
	for(i=0; i<domainTimeSlices-1; i++)
		AdvanceTime();
	cout << "FreeHeap after Domain allocated:" << farcoreleft() << "\n";
	}


Domain::~Domain()
	{
	POINT *tmp;
	int i;

	cout << "Destructing Domain class\n";
	if (domainHead == (POINT**)NULL)
		{
		cout << "Not freeing domain - DomainHead was NULL\n";
		return;
		}
	for(i=0; i<domainTimeSlices; i++)
		{
		tmp = domainHead[i];
		farfree(tmp);
		}
	farfree(domainHead);
	domainHead = (POINT**)NULL;
	cout << "FreeHeap after Domain deallocated:" << farcoreleft() << "\n";
//	getch();
	}


void Domain::AdvanceTime()
	{
	AgeTimeSlices();
//	DrawDomain();
	ClearVerticesInTimeSlice(domainTimeSlices-1);
	MoveMobileObjects();
//	DrawDomain();
	SetVerticesInTimeSlice(domainTimeSlices-1);
//	DrawDomain();
	}


///////////////////////////////////////
//
// Copy the contents of every timeslice
// into the previous timeslice
// [0] = [1], [1] = [2], etc.
// Leave the last timeslice unchanged.
// Another routine will decide the moves
// for all the mobile objects.
```

```
//
// A quick way to do this is moving ptrs
// to the timeslices and only copying the
// contents of the last timeslice over the
// contents of the first timeslice
//
////////////////////////////////////
void Domain::AgeTimeSlices()
        {
        int i, j;
        POINT *tmp;

        tmp = domainHead[0];
        for(i=0, j=1; i<domainTimeSlices-1; i++, j++)
                domainHead[i] = domainHead[j];
        domainHead[domainTimeSlices-1] = tmp;
        _fmemcpy(domainHead[domainTimeSlices-1],
                        domainHead[domainTimeSlices-2],
                        sizeof(*domainHead[domainTimeSlices-1]));
        }


int Domain::CalcRobotDir(int st, int sw, int sl, int et, int ew, int el)
   {
        if ((!IsPointStart(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(clear);

        if ((ew - sw > 0)
        && (el - sl > 0))
                {
                sw++, sl++;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
                || (IsPointGoal(st, sw, sl)))
                        return(frontright);
                else
                        return(clear);
                }

        if ((ew - sw > 0)
        && (el - sl == 0))
                {
                sw++;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
                || (IsPointGoal(st, sw, sl)))
                        return(right);
                else
                        return(clear);
                }

        if ((ew - sw > 0)
        && (el - sl < 0))
                {
                sw++, sl--;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
                || (IsPointGoal(st, sw, sl)))
                        return(backright);
                else
                        return(clear);
                }

        if ((ew - sw == 0)
        && (el - sl > 0))
                {
                sl++;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
                || (IsPointGoal(st, sw, sl)))
                        return(front);
                else
                        return(clear);
                }

        if ((ew - sw == 0)  // silly, but just in case
        && (el - sl == 0))
                {
                return(clear);
                }

        if ((ew - sw == 0)
        && (el - sl < 0))
```

```
		{
		sl--;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(back);
		else
			return(clear);
		}

	if ((ew - sw < 0)
	&& (el - sl > 0))
		{
		sw--, sl++;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(frontleft);
		else
			return(clear);
		}

	if ((ew - sw < 0)
	&& (el - sl == 0))
		{
		sw--;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(left);
		else
			return(clear);
		}

	if ((ew - sw < 0)
	&& (el - sl < 0))
		{
		sw--, sl--;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(backleft);
		else
			return(clear);
		}
	return(clear);
	}


float Domain::CheckLine(int st, int sw, int sl, int et, int ew, int el)
	{
//	static int displayCounter=1;
	int tmpt, tmpw, tmpl;
	float dist;

	for(dist=0.0, tmpt=et-st, tmpw=ew-sw, tmpl=el-sl;
		(tmpt != 0) || (tmpw != 0) || (tmpl != 0);
		tmpt=et-st, tmpw=ew-sw, tmpl=el-sl)
		{
		if (tmpt < 0)
			{
			return(-1.0);
			}
		if ((tmpw > 0)
		&& (tmpl > 0))
			{
			dist += GetPointCost(st, sw++, sl++, frontright);
			if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
	if ((tmpw > 0)
		&& (tmpl == 0))
			{
	  dist += GetPointCost(st, sw++, sl, right);
		if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
			}
		if ((tmpw > 0)
		&& (tmpl < 0))
```

```
			{
		dist += GetPointCost(st, sw++, sl--, backright);
			if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
		continue;
		}
		if ((tmpw < 0)
		&& (tmpl > 0))
			{
		dist += GetPointCost(st, sw--, sl++, frontleft);
		if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
		continue;
		}
		if ((tmpw < 0)
		&& (tmpl == 0))
			{
		dist += GetPointCost(st, sw--, sl, left);
		if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
		continue;
		}
		if ((tmpw < 0)
		&& (tmpl < 0))
			{
		dist += GetPointCost(st, sw--, sl--, backleft);
			if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
		continue;
			}
		if ((tmpw == 0)
		&& (tmpl > 0))
			{
		dist += GetPointCost(st, sw, sl++, front);
		if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
		continue;
		}
		if ((tmpw == 0)
		&& (tmpl < 0))
			{
		dist += GetPointCost(st, sw, sl--, back);
		if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
		continue;
		}
	if ((tmpw == 0)
	&& (tmpl == 0)
	&& (tmpt >0))
			{
		dist += GetPointCost(st++, sw, sl, up);
		if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
		continue;
		}
		}
//	cout << "CheckLine:" << displayCounter++ << ":returned" << dist << "\n";
	return(dist);
	}


int Domain::ClearAdjPointOK(int t, int w, int l)
	{
	char type;
	int a,b;

	for(a=w-1;a<=w+1;a++)
		{
		if ((a < 0) || (a >= domainWidth))
			continue;
		for(b=l-1;b<=l+1;b++)
			{
			if ((b < 0) || (b >= domainLength))
				continue;
			if ((a == w) && (b == l))
				continue;
```

```
          type = GetPointType(t, a, b);
                        if ((type == OBJECT)
                        ||  (type == MOBILE_OBJECT))
                              return(FALSE);
                        }

                }
        return(TRUE);
        }


void Domain::ClearMobileObject(int t, int w, int l)
    {
        int i, j;

        if (GetPointType(t, w, l) == MOBILE_OBJECT)
                {
                if (ClearAdjPointOK(t, w, l))
                        {
                        SetPointType(t, w, l, CLEAR);
                        SetPointFrom(t, w, l, FROM_NOWHERE);
                        }
                else
                        {
                        SetPointType(t, w, l, ADJ_TO_OBJECT);
                        SetPointFrom(t, w, l, FROM_NOWHERE);
                        }
                }
        for(i=w-1, j=l-1; j<l+2;)
                {
        if ((GetPointType(t, i, j) == ADJ_TO_OBJECT)
                && (ClearAdjPointOK(t, i, j)))
                {
                SetPointType(t, i, j, CLEAR);
                        SetPointFrom(t, i, j, FROM_NOWHERE);
                        }
                if (i == w+1)
                        {
                        i = w-1;
                        j++;
                        }
                else
                        i++;
                }
        }


void Domain::ClearVerticesInTimeSlice(int t)
        {
        int w, l;

        for(w=0; w<domainWidth; w++)
                {
                for(l=0; l<domainLength; l++)
                        {
                        if (GetPointType(t, w, l) == VERTEX)
            {
                                SetPointFrom(t, w, l, FROM_NOWHERE);
                                SetPointType(t, w, l, CLEAR);
            continue;
            }
                        if (GetPointType(t, w, l) == MOBILE_OBJECT)
                                {
            ClearMobileObject(t, w, l);
                                }
                        }
                }
        }


void Domain::DrawDomain()
        {
        int i;

        for (i=0; i< domainTimeSlices; i++)
                {
                DrawTimeSlice(i);
//              getch();
                }
        }


void Domain::DrawTimeSlice(int timeSlice)
        {
        char type;
```

```
        int a,b,y;

        clrscr();
        gotoxy(1,1);
        cout << "time=t+" << timeSlice;

        for(b=0, y=2;b<domainLength; b++, y++)
                {
                gotoxy(1, y);
                for(a=0; a<domainWidth; a++)
                        {
                        type = GetPointType(timeSlice, a, b);
                        switch(type)
                                {
                                case GOAL:
                                case START:
                                case ADJ_TO_OBJECT:
                                case OBJECT:
                                case MOBILE_OBJECT:
                                case VERTEX:
                                        cout << type;
                                        break;
                                case CLEAR:
                                        switch(GetPointFrom(timeSlice, a, b))
                                                {
                                                case FROM_RIGHT:
                                                        cout << "R";
                                                        break;
                                                case FROM_LEFT:
                                                        cout << "L";
                                                        break;
                                                case FROM_UP:
                                                        cout << "U";
                                                        break;
                                                case FROM_DOWN:
                                                        cout << "D";
                                                        break;
                                                case FROM_FRONT:
                                                        cout << "F";
                                                        break;
                                                case FROM_BACK:
                                                        cout << "B";
                                                        break;
                                                case FROM_BACKLEFT:
                                                        cout << "T";
                                                        break;
                                                case FROM_BACKRIGHT:
                                                        cout << "U";
                                                        break;
                                                case FROM_FRONTLEFT:
                                                        cout << "V";
                                                        break;
                                                case FROM_FRONTRIGHT:
                                                        cout << "W";
                                                        break;
                                                case FROM_NOWHERE:
                                                default:
                                                        cout << ".";
                                                        break;
                                                }
                                        break;
                                default:
                                        cout << "?";
                                        break;
                                }
                        }
                cout << "\n";
                }
        }


POINT* Domain::GetPoint(int timeSlice, int width, int length)
        {
   POINT *tmp, *tmp2;

   tmp = domainHead[timeSlice];
      tmp2 = tmp + (domainWidth*width) + length;
   return(tmp2);
   }


float Domain::GetPointCost(int timeSlice, int width, int length, int dir)
   {
   POINT *tmp;
```

```
    tmp = GetPoint(timeSlice, width, length);
        return(tmp->cost[dir]);
    }

char Domain::GetPointFrom(int timeSlice, int width, int length)
    {
    POINT *tmp;

    tmp = GetPoint(timeSlice, width, length);
        return(tmp->from);
    }


char Domain::GetPointType(int timeSlice, int width, int length)
        {
    POINT *tmp;

        tmp = GetPoint(timeSlice, width, length);
        return(tmp->type);
        }


void Domain::InitTimeSlice(int timeSlice)
        {
        POINT*tmp;
    int a,b;

    for(a=0;a<domainWidth;a++)
                {
                for(b=0;b<domainLength;b++)
                        {
        tmp = GetPoint(timeSlice, a, b);
                        tmp->dist=0.0;
                        tmp->from=FROM_NOWHERE;
                        tmp->type=CLEAR;

                        if (timeSlice == domainTimeSlices-1)
                                tmp->cost[FROM_UP] = BLOCKED;
                        else
                                tmp->cost[FROM_UP] = NORMAL;

                        if ((a == 0) && (b == 0))
                                {
                                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                                tmp->cost[FROM_RIGHT] = NORMAL;
                                tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                tmp->cost[FROM_BACK] = BLOCKED;
                                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                                tmp->cost[FROM_LEFT] = BLOCKED;
                                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                                tmp->cost[FROM_FRONT] = NORMAL;
                                continue;
                                }
                        if ((a == 0) && (b < domainLength-1))
                                {
                                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                                tmp->cost[FROM_RIGHT] = NORMAL;
                                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                                tmp->cost[FROM_BACK] = NORMAL;
                                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                                tmp->cost[FROM_LEFT] = BLOCKED;
                                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                                tmp->cost[FROM_FRONT] = NORMAL;
                                continue;
                                }
                        if ((a == 0) && (b==domainLength-1))
                                {
                                tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                                tmp->cost[FROM_RIGHT] = NORMAL;
                                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                                tmp->cost[FROM_BACK] = NORMAL;
                                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                                tmp->cost[FROM_LEFT] = BLOCKED;
                                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                                tmp->cost[FROM_FRONT] = BLOCKED;
                                continue;
                                }
                        if ((a < domainWidth-1) && (b == 0))
                                {
                                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                                tmp->cost[FROM_RIGHT] = NORMAL;
                                tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                tmp->cost[FROM_BACK] = BLOCKED;
```

```
                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                tmp->cost[FROM_LEFT] = NORMAL;
                tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
        if ((a < domainLength-1) && (b < domainWidth-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_LEFT] = NORMAL;
                tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
        if ((a < domainWidth-1) && (b == domainLength-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_LEFT] = NORMAL;
                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                tmp->cost[FROM_FRONT] = BLOCKED;
                continue;
                }
        if ((a == domainWidth-1) && (b == 0))
                {
                tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                tmp->cost[FROM_RIGHT] = BLOCKED;
                tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                tmp->cost[FROM_BACK] = BLOCKED;
                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                tmp->cost[FROM_LEFT] = NORMAL;
                tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
        if ((a == domainWidth-1) && (b < domainWidth-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                tmp->cost[FROM_RIGHT] = BLOCKED;
                tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_LEFT] = NORMAL;
                tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
        if ((a == domainLength-1) && (b == domainWidth-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                tmp->cost[FROM_RIGHT] = BLOCKED;
                tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_LEFT] = NORMAL;
                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                tmp->cost[FROM_FRONT] = BLOCKED;
                continue;
                }
        }
    }
}


int Domain::IsPointClear(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
    || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case CLEAR:
```

```
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointGoal(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
    || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case GOAL:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }



int Domain::IsPointNearObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
                case OBJECT:
                        return(OBJECT);
                case MOBILE_OBJECT:
                        return(MOBILE_OBJECT);
                case ADJ_TO_OBJECT:
                        return(ADJ_TO_OBJECT);
                default:
                        return(FALSE);
                }
    }

int Domain::IsPointObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
                case OBJECT:
                        return(OBJECT);
                case MOBILE_OBJECT:
                        return(MOBILE_OBJECT);
                case ADJ_TO_OBJECT:
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointStart(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
```

```
		{
		case START:
			return(TRUE);
		default:
			return(FALSE);
		}
	}


int Domain::IsPointVertex(int t, int w, int l)
	{
	if ((w < 0)
	|| (l < 0)
	|| (t < 0)
	|| (w >= domainWidth)
	|| (l >= domainLength)
	|| (t >= domainTimeSlices))
		return(FALSE);
	switch(GetPointType(t, w, l))
		{
		case VERTEX:
			return(TRUE);
		default:
			return(FALSE);
		}
	}


void Domain::MarkMobileObject(int t, int w, int l)
	{
	int i, j;

	if ((GetPointType(t, w, l) == CLEAR)
	|| (GetPointType(t, w, l) == ADJ_TO_OBJECT))
		{
		SetPointType(t, w, l, MOBILE_OBJECT);
		SetPointFrom(t, w, l, FROM_NOWHERE);
		}
	for(i=w-1, j=l-1; j<l+2;)
		{
	if (GetPointType(t, i, j) == CLEAR)
		{
		SetPointType(t, i, j, ADJ_TO_OBJECT);
		SetPointFrom(t, i, j, FROM_NOWHERE);
		}
	if (i == w+1)
	  {
	  i = w-1;
	  j++;
	  }
	else
			i++;
		}
	}


void Domain::MoveMobileObject(OBJECT_NODE*object)
	{
	int t=domainTimeSlices-1, w=(*object).w, l=(*object).l;

	ClearMobileObject(t, w, l);
	switch((*object).direction)
		{
		case front:
			if (IsPointObject(t, w, l+1))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).l++;
			break;
		case frontleft:
			if (IsPointObject(t, w-1, l+1))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).l++, (*object).w--;
			break;
		case left:
			if (IsPointObject(t, w-1, l))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w--;
			break;
		case backleft:
			if (IsPointObject(t, w-1, l-1))
				(*object).direction = rand() % NUM_DIRS;
```

```
                    else
                            (*object).w--, (*object).l--;
                    break;
            case back:
                    if (IsPointObject(t, w, l-1))
                            (*object).direction = rand() % NUM_DIRS;
                    else
                            (*object).l--;
                    break;
            case backright:
                    if (IsPointObject(t, w+1, l-1))
                            (*object).direction = rand() % NUM_DIRS;
                    else
                            (*object).w++, (*object).l--;
                    break;
            case right:
                    if (IsPointObject(t, w+1, l))
                            (*object).direction = rand() % NUM_DIRS;
                    else
                            (*object).w++;
                    break;
            case frontright:
                    if (IsPointObject(t, w+1, l+1))
                            (*object).direction = rand() % NUM_DIRS;
                    else
                            (*object).w++, (*object).l++;
                    break;
            default:
                    cout << "*** unknown direction for mobile object ignored ***\n";
                    break;
            }
        MarkMobileObject(t, object->w, object->l);
        }


void Domain::MoveMobileObjects()
        {
        OBJECT_NODE*cur, *orig;

        orig = cur = objList.GetNextObject();
        if (cur == (OBJECT_NODE*)NULL)
       return;

        do
                {
                if ((*cur).velocity > 0)
                        MoveMobileObject(cur);

       cur = objList.GetNextObject();
                } while (cur != orig);
        }


float Domain::MoveRobot(int st, int sw, int sl, int et, int ew, int el)
        {
        float dist = 0.0;
        int dir;

        if (!IsPointStart(st, sw, sl))
                return(dist);

        dir = CalcRobotDir(st, sw, sl, et, ew, el);
        SetPointType(st, sw, sl, CLEAR);
        SetPointFrom(st, sw, sl, FROM_NOWHERE);

        switch(dir)
                {
                case frontright:
                        dist = GetPointCost(st, sw, sl, frontright);
                        st++, sw++, sl++;
                        if ((!IsPointClear(st, sw, sl))
                        && (!IsPointVertex(st, sw, sl))
                        && (!IsPointGoal(st, sw, sl)))
                                {
                                st--, sw--, sl--;
                                dist = 0.0;
                                }
                        break;
                case right:
                        dist = GetPointCost(st, sw, sl, right);
                        st++, sw++;
                        if ((!IsPointClear(st, sw, sl))
                        && (!IsPointVertex(st, sw, sl))
```

```
        && (!IsPointGoal(st, sw, sl)))
                {
                st--, sw--;
                dist = 0.0;
                }
        break;
case backright:
        dist = GetPointCost(st, sw, sl, backright);
        st++, sw++, sl--;
        if ((!IsPointClear(st, sw, sl))
        && (!IsPointVertex(st, sw, sl))
        && (!IsPointGoal(st, sw, sl)))
                {
                st--, sw--, sl++;
                dist = 0.0;
                }
        break;
case frontleft:
        dist = GetPointCost(st, sw, sl, frontleft);
        st++, sw--, sl++;
        if ((!IsPointClear(st, sw, sl))
        && (!IsPointVertex(st, sw, sl))
        && (!IsPointGoal(st, sw, sl)))
                {
                st--, sw++, sl--;
                dist = 0.0;
                }
        break;
case left:
        dist = GetPointCost(st, sw, sl, left);
        st++, sw--;
        if ((!IsPointClear(st, sw, sl))
        && (!IsPointVertex(st, sw, sl))
        && (!IsPointGoal(st, sw, sl)))
                {
                st--, sw++;
                dist = 0.0;
                }
        break;
case backleft:
        dist = GetPointCost(st, sw, sl, backleft);
        st++, sw--, sl--;
        if ((!IsPointClear(st, sw, sl))
        && (!IsPointVertex(st, sw, sl))
        && (!IsPointGoal(st, sw, sl)))
                {
                st--, sw++, sl++;
                dist = 0.0;
                }
        break;
case front:
        dist = GetPointCost(st, sw, sl, front);
        st++, sl++;
        if ((!IsPointClear(st, sw, sl))
        && (!IsPointVertex(st, sw, sl))
        && (!IsPointGoal(st, sw, sl)))
                {
                st--, sl--;
                dist = 0.0;
                }
        break;
case back:
        dist = GetPointCost(st, sw, sl, back);
        st++, sl--;
        if ((!IsPointClear(st, sw, sl))
        && (!IsPointVertex(st, sw, sl))
        && (!IsPointGoal(st, sw, sl)))
                {
                st--, sl++;
                dist = 0.0;
                }
        break;
case up:
        dist = GetPointCost(st, sw, sl, up);
        st++;
        if ((!IsPointClear(st, sw, sl))
        && (!IsPointVertex(st, sw, sl))
        && (!IsPointGoal(st, sw, sl)))
                {
                st--;
                dist = 0.0;
                }
        break;
default:
```

```
                break;
            }
        SetPointType(st, sw, sl, START);
        SetPointFrom(st, sw, sl, FROM_NOWHERE);
        return(dist);
        }


void Domain::SetAdjObjsInTimeSlice(int timeSlice)
        {
        int a,b;
        int w,l;

        for(w=0; w<domainWidth; w++)
            {
            for(l=0; l<domainLength; l++)
                {
                if (!IsPointObject(timeSlice, w, l))
                    continue;

                for(a=w-1; a<=w+1; a++)
                    {
                    if ((a < 0) || (a >= domainWidth))
                        continue;
                    for(b=l-1; b<=l+1; b++)
                        {
                        if ((b < 0) || (b >= domainLength))
                            continue;

                        if (GetPointType(timeSlice, a, b) == CLEAR)
                            {
                            SetPointFrom(timeSlice, a, b, FROM_NOWHERE);
                            SetPointType(timeSlice, a, b, ADJ_TO_OBJECT);
                            }

                        }
                    }
                }
            }
        }


void Domain::SetGoalFromFile(char*fileName)
        {
        int i, w, l;
        char recType;
        char tmp[256];
        fstream dataFile;

        dataFile.open(fileName, ios::in);
        if (!dataFile)
            cout << "Unable to open " << fileName << ". No Goal added.\n";

        for(;;)
            {
            dataFile. getline(tmp, sizeof(tmp));
            if (strlen(tmp) == 0)
                break;
            if (tmp[0] == GOAL)
                {
//              cout << "The GOAL line is: " << tmp << "\n";
                if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                    {
          for(i=0; i<domainTimeSlices; i++)
              {
                    SetPointFrom(i, w, l, FROM_NOWHERE);
                    SetPointType(i, w, l, GOAL);
              }
                    break;
                    }
                else
                    cout << "Improperly formatted line ignored\n";
                }
            }
        dataFile.close();
        }


void Domain::SetMobileObjsFromFile(char*fileName)
        {
        int w, l;
        char recType;
        char tmp[256];
        fstream dataFile;
        OBJECT_NODE* newObj;
```

```
        dataFile.open(fileName, ios::in);
        if (!dataFile)
                cout << "Unable to open " << fileName << ". No Mobile objs added.\n";

        for(;;)
                {
                int i;

                dataFile. getline(tmp, sizeof(tmp));
                i = strlen(tmp);
                if (i <= 0)
                        break;
                cout << "Length of input line is: " << i << "\n";
                if ((tmp[0] != GOAL)
                && (tmp[0] != START))
                        {
//                      cout << "OBJECT line is: " << tmp << "\n";
                        if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                                {
                                SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
                                SetPointType(domainTimeSlices-1, w, l, MOBILE_OBJECT);
                                newObj = objList.BuildNewObject(w, l);
                                if (newObj)
                                        {
                                        objList.InsertNewObject(newObj);
//                                      cout << "Added obj to obj list\n";
                                        }
                                }
                        else
                                cout << "Improperly formatted line ignored\n";
                        }
                }
        dataFile.close();
        }


void Domain::SetPermObjsInTimeSlice(int timeSlice)
        {
        int w, l;

        for(w=0; w<domainWidth-5; w++)
                {
                l=2;
                switch(w)
                        {
                        case 3:
                        case 4:
                        case 5:
                                break;
                        default:
                                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                                SetPointType(timeSlice, w, l, OBJECT);
//                              SetAdjObjsInTimeSlice(timeSlice, w, l);
                                break;
                        }
                }
        for(w=0; w<domainWidth-10; w++)
                {
                l=7;
                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                SetPointType(timeSlice, w, l, OBJECT);
//              SetAdjObjsInTimeSlice(timeSlice, w, l);
                }
        for(l=0, w=domainWidth-10; l<domainLength-2; l++)
                {
                switch(l)
                        {
                        case 3:
                        case 4:
                        case 5:
                        case 6:
                        case 14:
                        case 15:
                        case 16:
                                break;
                        default:
                                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                                SetPointType(timeSlice, w, l, OBJECT);
//                              SetAdjObjsInTimeSlice(timeSlice, w, l);
                                break;
                        }
                }
        for(l=2, w=domainWidth-5; l<domainLength-7; l++)
```

```
		{
		switch(l)
			{
			case 9:
			case 10:
			case 11:
				break;
			default:
				SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
				SetPointType(timeSlice, w, l, OBJECT);
//				SetAdjObjsInTimeSlice(timeSlice, w, l);
				break;
			}
		}
	for(w=domainWidth-5, l=domainLength-7; w<domainWidth; w++)
		{
		SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
		SetPointType(timeSlice, w, l, OBJECT);
//		SetAdjObjsInTimeSlice(timeSlice, w, l);
		}
	for(w=domainWidth-10, l=domainLength-2; w<domainWidth; w++)
		{
		SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
		SetPointType(timeSlice, w, l, OBJECT);
//		SetAdjObjsInTimeSlice(timeSlice, w, l);
		}
	}


void Domain::SetPointFrom(int timeSlice, int width, int length, int from)
	{
	POINT *tmp;

	if ((width < 0)
	|| (length < 0)
	|| (timeSlice < 0)
	|| (width >= domainWidth)
	|| (length >= domainLength)
	|| (timeSlice >= domainTimeSlices))
		return;
	tmp = GetPoint(timeSlice, width, length);
	tmp->from = from;
	}

void Domain::SetPointType(int timeSlice, int width, int length, char type)
	{
	char oldType;
	POINT *tmp;

	if ((width < 0)
	|| (length < 0)
	|| (timeSlice < 0)
	|| (width >= domainWidth)
	|| (length >= domainLength)
	|| (timeSlice >= domainTimeSlices))
		return;
	tmp = GetPoint(timeSlice, width, length);
	oldType = tmp->type;
	tmp->type = type;
	if (oldType != OBJECT)
		{
//		SetAdjObjsInTimeSlice(timeSlice, width, length);
		}
	}


void Domain::SetStartFromFile(char*fileName)
	{
	int w, l;
	char recType;
	char tmp[256];
	fstream dataFile;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Start added.\n";

	for(;;)
		{
		dataFile. getline(tmp, sizeof(tmp));
		if (strlen(tmp) == 0)
			break;
		if (tmp[0] == START)
			{
```

```
//                      cout << "The START line is: " << tmp << "\n";
                        if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                                {
                                SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
                                SetPointType(domainTimeSlices-1, w, l, START);
                                break;
                                }
                        else
                                cout << "Improperly formatted line ignored\n";
                        }
                }
        dataFile.close();
        }


void Domain::SetVerticesInTimeSlice(int t)
        {
        int w, l;
        int diff_counter, corner_counter;
        int side_bits, corner_bits;

        for(w=0; w<domainWidth; w++)
                {
                for(l=0; l<domainLength; l++)
                        {
                        if (!IsPointClear(t, w, l))
                                continue;

                        corner_bits = BITMASK_CLEAR;
                        corner_counter = 0;
                        if (IsPointNearObject(t, w-1, l-1))
                                {
                                corner_bits |= BITMASK_LEFT;
                                corner_bits |= BITMASK_TOP;
                                corner_counter++;
                                }
                        if (IsPointNearObject(t, w-1, l+1))
                                {
                                corner_bits |= BITMASK_LEFT;
                                corner_bits |= BITMASK_BOTTOM;
                                corner_counter++;
                                }
                        if (IsPointNearObject(t, w+1, l+1))
                                {
                                corner_bits |= BITMASK_RIGHT;
                                corner_bits |= BITMASK_BOTTOM;
                                corner_counter++;
                                }
                        if (IsPointNearObject(t, w+1, l-1))
                                {
                                corner_bits |= BITMASK_RIGHT;
                                corner_bits |= BITMASK_TOP;
                                corner_counter++;
                                }
                        if (corner_bits != BITMASK_CLEAR)
                                {
                                side_bits = BITMASK_CLEAR;
                                if (IsPointNearObject(t, w-1, l))
                                        side_bits |= BITMASK_LEFT;
                                if (IsPointNearObject(t, w, l-1))
                                        side_bits |= BITMASK_TOP;
                                if (IsPointNearObject(t, w+1, l))
                                        side_bits |= BITMASK_RIGHT;
                                if (IsPointNearObject(t, w, l+1))
                                        side_bits |= BITMASK_BOTTOM;

                                for(diff_counter=0;
                                        (side_bits != BITMASK_CLEAR)
                                        || (corner_bits != BITMASK_CLEAR);
                                        side_bits >>= 1, corner_bits >>= 1)
                                        {
                                        if ((corner_bits & (int)0x01)
                                        && ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
                                                {
                                                diff_counter++;
                                                continue;
                                                }
                                        if ((side_bits & (int)0x01)
                                        && ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
                                                {
                                                diff_counter++;
                                                continue;
                                                }
                                        }
```

```
                                        if ((diff_counter > 2)
                                        || ((diff_counter == 2) && (corner_counter == 1)))
                                              {
                                              SetPointFrom(t, w, l, FROM_NOWHERE);
                                              SetPointType(t, w, l, VERTEX);
                                              }
                                        }
                                  }
                            }
                      }

// edge.hpp
typedef struct van
   {
        int t,w,l;
        float dist;
        struct van *prev, *next;
        } EDGE_NODE;

class EdgeList
        {
        public:
                EdgeList();
                ~EdgeList();
        EDGE_NODE* BuildNewEdge(int, int, int, float);
        void DelEdge(EDGE_NODE*);
                void DelEdgeToVertex(int, int, int);
        EDGE_NODE* GetFirstEdge(void);
        EDGE_NODE* GetNextEdge(EDGE_NODE*);
        void InsertNewEdge(EDGE_NODE*);
        void ListAllEdges(void);
   private:
        EDGE_NODE *edgeHead;
        };

// edge.cpp
#include <iostream.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include "edge.hpp"


EdgeList::EdgeList()
        {
        edgeHead = (EDGE_NODE*)NULL;
//      cout << "EdgeList Constructor\n";
        }


EdgeList::~EdgeList()
        {
        while (edgeHead != (EDGE_NODE*)NULL)
                DelEdge(edgeHead);
//      cout << "EdgeList Destructor\n";
        }


EDGE_NODE* EdgeList::BuildNewEdge(int t, int w, int l, float dist)
        {
        EDGE_NODE *a;

//      if (otherVertex != (void*)NULL)
//              {
                a = (EDGE_NODE*)malloc(sizeof(EDGE_NODE));
                if (a == (EDGE_NODE*)NULL)
                        {
                        cout << "Out of Memory in BuildNewEdge()\n";
                        exit(0);
                        }
                (*a).t = t;
                (*a).w = w;
                (*a).l = l;
                (*a).dist = dist;
                (*a).prev = (*a).next = (EDGE_NODE*)NULL;
                return(a);
//              }
//      return((EDGE_NODE*)NULL);
        }


void EdgeList::DelEdge(EDGE_NODE*a)
   {
   EDGE_NODE*tmp;
```

```
//      cout << "edge 0\n";
        if (a != (EDGE_NODE*)NULL)
                {
                if ((a->prev == (EDGE_NODE*)NULL)  // del last remaining edge
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 1\n";
                        free(a);
                        edgeHead = (EDGE_NODE*)NULL;
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge in middle
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 2\n";
                        tmp = a->prev;
                        tmp->next = a->next;
                        tmp = a->next;
                        tmp->prev = a->prev;
                        free(a);
                        return;
                        }
                if ((a->prev == (EDGE_NODE*)NULL)  // del edge at sol
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 3\n";
                        tmp = a->next;
                        tmp->prev = (EDGE_NODE*)NULL;
                        edgeHead = tmp;
                        free(a);
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge at eol
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 4\n";
                        tmp = a->prev;
                        tmp->next = (EDGE_NODE*)NULL;
                        free(a);
                        return;
                        }
                }
        }


void EdgeList::DelEdgeToVertex(int t, int w, int l)
        {
        EDGE_NODE*tmp;

        for(tmp = GetFirstEdge(); tmp != (EDGE_NODE*)NULL; tmp = GetNextEdge(tmp))
                {
                if ((tmp->t == t)
        && (tmp->w == w)
        && (tmp->l == l))
                        {
                        DelEdge(tmp);
                        break;
                        }
                }
        }


EDGE_NODE* EdgeList::GetFirstEdge()
        {
   return(edgeHead);
   }


EDGE_NODE* EdgeList::GetNextEdge(EDGE_NODE*cur)
   {
   return(cur->next);
   }


void EdgeList::InsertNewEdge(EDGE_NODE *edge)
        {
        EDGE_NODE*cur;

        if (edge == (EDGE_NODE*)NULL)
                return;

        if (edgeHead == (EDGE_NODE*)NULL)
                {
```

```
		edgeHead = edge;
		edge->prev = edge->next = (EDGE_NODE*)NULL;
//		cout << "Inserted edge into empty list ";
//		cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//		cout << "Dist = " << edge->dist << "\n";
		return;
		}

	for(cur=edgeHead; cur != (EDGE_NODE*)NULL; cur=(*cur).next)
		{
		if ( (edge->t > cur->t)
		|| ((edge->t == cur->t)
			&& (edge->w > cur->w))
		|| ((edge->t == cur->t)
			&& (edge->w == cur->w)
			&& (edge->l > cur->l)) )
			{				// insert after cur
			if (cur->next == (EDGE_NODE*)NULL)
				{
				edge->next = (EDGE_NODE*)NULL;// no more so append to eol
				edge->prev = cur;
				cur->next = edge;
				break;
				}
			else
				continue;			// try next one
			}
		if ( (edge->t < cur->t)
		|| ((edge->t == cur->t)
			&& (edge->w < cur->w))
		|| ((edge->t < cur->t)
			&& (edge->w == cur->w)
			&& (edge->l < cur->l)) )
			{				// insert before cur
			if (cur->prev == (EDGE_NODE*)NULL)
				{
				edge->prev = (EDGE_NODE*)NULL;	// at start of list
				edge->next = cur;
				cur->prev = edge;
				edgeHead = edge;
				break;
				}
			else
				{
				edge->prev = cur->prev;		// in middle/end of list
				edge->next = cur;
				cur->prev = edge;
				cur = edge->prev;
				cur->next = edge;
				break;
				}
			}
		if ( (edge->t == cur->t)
		&& (edge->w == cur->w)
		&& (edge->l == cur->l) )
			{
			/* already here - replace it ! */
			cout << "Already here - replacing values and disposing of new edge ";
	cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
	cout << "Dist = " << edge->dist << "\n";
			cur->w = edge->w;
			cur->l = edge->l;
			cur->dist = edge->dist;
			free(edge);
			break;
			}
		}
//	cout << "Inserted edge into list ";
//	cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//	cout << "Dist = " << edge->dist << "\n";
	}


void EdgeList::ListAllEdges()
	{
	EDGE_NODE *edge;

	for(edge = edgeHead; edge != (EDGE_NODE*)NULL; edge = (*edge).next)
		{
		cout << "Edge to (";
		cout << (*edge).t << ",";
		cout << (*edge).w << ",";
		cout << (*edge).l << "). Dist is " << (*edge).dist << "\n";
```

```
        }
        cout << "-----------------\n";
        }

// object.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define NUM_DIRS 7 /* this is the 8 horizontal directions; (0->7) */
enum directions { front, back, left, right, frontright, frontleft, backright, backleft, up, down, clear};


typedef struct o
        {
        int direction, velocity;
        int w, l;
        struct o *prev, *next;
        } OBJECT_NODE;


class ObjectList
        {
        public:
                ObjectList();
                ~ObjectList();
                OBJECT_NODE* BuildNewObject(int, int);
        void DelAllObjects(void);
        void DelObject(OBJECT_NODE*);
        OBJECT_NODE* GetNextObject(void);
                void InsertNewObject(OBJECT_NODE*);
        void ListAllObjects(void);
        private:
                OBJECT_NODE *objectHead;
        };

// object.cpp
#include <iostream.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
// #include "edge.hpp"
#include "object.hpp"

ObjectList::ObjectList()
        {
        objectHead = (OBJECT_NODE*)NULL;
        cout << "ObjectList Constructor\n";
        }



ObjectList::~ObjectList()
        {
        cout << "ObjectList Destructor\n";
        DelAllObjects();
        }



OBJECT_NODE* ObjectList::BuildNewObject(int w, int l)
        {
        OBJECT_NODE *newPtr;

        newPtr = (OBJECT_NODE*)malloc(sizeof(OBJECT_NODE));
        if (newPtr == (OBJECT_NODE*)NULL)
                {
                cout << "Out of memory in BuildNewObject()\n";
                return(NULL);
                }
        (*newPtr).direction = rand() % NUM_DIRS;
        (*newPtr).velocity = 1;
        (*newPtr).prev = (*newPtr).next = (OBJECT_NODE*)NULL;
        (*newPtr).w = w;
        (*newPtr).l = l;
        return(newPtr);
        }
```

```
void ObjectList::DelAllObjects()
	{
	OBJECT_NODE *tmp;

	while (objectHead != (OBJECT_NODE*)NULL)
		{
		tmp = objectHead;
		objectHead = (*objectHead).next;
		DelObject(tmp);
		}
	}


void ObjectList::DelObject(OBJECT_NODE *todie)
  {
	OBJECT_NODE *cur;

	if (((*todie).prev != (OBJECT_NODE*)NULL)
	&& ((*todie).next != (OBJECT_NODE*)NULL))
		{
//		cout << "Deleted Object (" << todie->w << "," << todie->l << ")\n";
		cur=(*todie).prev;
		if (cur == todie)
			{
			objectHead = (OBJECT_NODE*)NULL;
//			cout << "Object List Empty\n";
			free(todie);
			return;
			}
		else
			{
			(*cur).next=(*todie).next;
			cur = (*todie).next;
			(*cur).prev = (*todie).prev;
			if (objectHead == todie)
				objectHead = (*todie).next;
	free(todie);
	  return;
			}
		}
	cout << "***Did NOT delete rotton Object (" << todie->w << "," << todie->l << ")\n";
	}


OBJECT_NODE* ObjectList::GetNextObject()
  {
  OBJECT_NODE*tmp;

	if (objectHead == (OBJECT_NODE*)NULL)
		return((OBJECT_NODE*)NULL);

	tmp = objectHead;
  objectHead = objectHead->next;
  return(tmp);
  }


void ObjectList::InsertNewObject(OBJECT_NODE*newPtr)
	{
	OBJECT_NODE *cur;

	if (newPtr == (OBJECT_NODE*)NULL)
		return;

	if (objectHead == (OBJECT_NODE*)NULL)
		{
		objectHead = newPtr;
		(*newPtr).prev = (*newPtr).next = newPtr;
		}
	else
		{
		/* insert before first node */
		cur = objectHead;
		(*newPtr).next = cur;
		(*newPtr).prev = (*cur).prev;
		(*cur).prev = newPtr;

		cur = (*newPtr).prev;
		(*cur).next = newPtr;
		objectHead = newPtr;
```

```
        }
//      cout << "Inserted object (" << newPtr->w << "," << newPtr->l << ")\n";
        }


void ObjectList::ListAllObjects()
        {
    OBJECT_NODE*tmp;

    if (objectHead == (OBJECT_NODE*)NULL)
        {
        cout << "ObjectList is empty\n";
        return;
        }
    for(tmp=objectHead;;tmp = tmp->next)
        {
         cout << "Object:(" << tmp->w << "," << tmp->l << ")\n";
        tmp = tmp->next;
        if (tmp == objectHead)
            {
            cout << "ObjectList ended\n";
            break;
            }
        }
        }

// vertex.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00


typedef struct vn
        {
        int t,w,l;
        char nodeType;
        struct vn *searchPrev, *searchNext;
        float searchDist;
        int searchMarker;
        EdgeList *edgeList;
        struct vn *pathFrom, *pathTo;
        struct vn *prev, *next;
        } VERTEX_NODE;


class VertexList
        {
        public:
                VertexList();
                ~VertexList();
                void AddToSearchList(VERTEX_NODE*);
        VERTEX_NODE* BuildNewVertex(int,int,int,char);
                int CalcRobotDir(VERTEX_NODE*);
                void DelAllVertices(void);
        void DelVertex(VERTEX_NODE*);
        int FindPath(void);
        VERTEX_NODE* FindVertex(int, int, int);
                VERTEX_NODE* GetFirstVertex(void);
                VERTEX_NODE* GetNextVertex(VERTEX_NODE*);
        VERTEX_NODE* GetStartVertex(void);
        void InsertAllVertices(void);
                void InsertNewVertex(VERTEX_NODE*);
        void ListAllVertices(void);
        void ListSearchList(void);
//              void MoveRobot(int, int, int, int);
                void MarkPath(VERTEX_NODE*);
                void RemoveFromSearchList(VERTEX_NODE*);
                void TrimSearchList(void);
        private:
                VERTEX_NODE* vertexHead;
```

```
       VERTEX_NODE* searchHead;
       int searchMarker;
       float searchTrimDist;
        };

// vertex.cpp
#include <iostream.h>
#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include "edge.hpp"
#include "vertex.hpp"

VertexList::VertexList()
        {
        vertexHead = (VERTEX_NODE*)NULL;
   searchHead = (VERTEX_NODE*)NULL;
   searchMarker = 0;
   searchTrimDist = -1.0;
   cout << "Initialised Vertex class\n";
        }



VertexList::~VertexList()
        {
        cout << "VertexList destructor started\n";
        DelAllVertices();
        cout << "VertexList destructor ended!\n";
        }


void VertexList::AddToSearchList(VERTEX_NODE*a)
        {
        VERTEX_NODE *tmp;

        if (a == (VERTEX_NODE*)NULL)
                return;

        if (((*a).searchPrev != (VERTEX_NODE*)NULL) // if already in fringe
        || ((*a).searchNext != (VERTEX_NODE*)NULL)) // list, dont add again
                return;

        if (searchHead == (VERTEX_NODE*)NULL)
                {
                searchHead = a;
                a->pathFrom = a->pathTo = (VERTEX_NODE*)NULL;
                a->searchPrev = a->searchNext = a;

//              cout << "Inserted into empty search list (";
//              cout << a->t << "," << a->w << "," << a->l << ")\n";
                }
        else
                {
                a->searchNext = searchHead;
                a->searchPrev = searchHead->searchPrev;
                searchHead->searchPrev = a;
                tmp = a->searchPrev;
                tmp->searchNext = a;
                searchHead = a;
//              cout << "Appended to search list (";
//              cout << a->t << "," << a->w << "," << a->l << ")\n";
                }
        }


VERTEX_NODE* VertexList::BuildNewVertex(int t, int w, int l, char nodeType)
        {
        VERTEX_NODE *newPtr;

        newPtr = (VERTEX_NODE*)malloc(sizeof(VERTEX_NODE));
        if (newPtr == (VERTEX_NODE*)NULL)
                {
                cout << "Malloc() failed in BuildNewVertex!\n";
                exit(0);
                }
        (*newPtr).t = t;
        (*newPtr).w = w;
        (*newPtr).l = l;
        (*newPtr).nodeType = nodeType;
        (*newPtr).edgeList = new EdgeList();
        (*newPtr).pathTo = (*newPtr).pathFrom = (VERTEX_NODE*)NULL;
        (*newPtr).searchDist = 0.0;
```

```
        (*newPtr).searchPrev = (*newPtr).searchNext = (VERTEX_NODE*)NULL;
        (*newPtr).prev = (*newPtr).next = (VERTEX_NODE*)NULL;
        if (nodeType == START)
                {
                (*newPtr).searchMarker = searchMarker+1;
                AddToSearchList(newPtr);
                }
        else
                (*newPtr).searchMarker = searchMarker;
        return(newPtr);
        }


void VertexList::DelAllVertices()
   {
        while (vertexHead != (VERTEX_NODE*)NULL)
                {
//              cout << "About to delete (" << vertexHead->t << "," << vertexHead->w ;
//              cout << "," << vertexHead->l << ") at addr: " << vertexHead ;
//              cout << ": Prev:" << vertexHead->prev << ": Next:" << vertexHead->next <<"\n";
                DelVertex(vertexHead);
                }
        cout << "----------\n";
//      getch();
        }



void VertexList::DelVertex(VERTEX_NODE *todie)
        {
        VERTEX_NODE *tmp;
        EDGE_NODE*a;

        if (todie == (VERTEX_NODE*)NULL)
                return;

        for(a = todie->edgeList->GetFirstEdge();
                a != (EDGE_NODE*)NULL;
                a = todie->edgeList->GetFirstEdge())
                {
       todie->edgeList->DelEdgeToVertex(a->t, a->w, a->l);
       tmp = FindVertex(a->t, a->w, a->l);
       if (tmp != (VERTEX_NODE*)NULL)
          {
          tmp->edgeList->DelEdgeToVertex(todie->t, todie->w, todie->l);
          }
//              tmp = (VERTEX_NODE*)(a->otherVertex);
//              cout << "On vertex (" << todie->t << "," << todie->w << ",";
//              cout << todie->l << "):Del edge to (" << tmp->t << "," << tmp->w << ",";
//              cout << tmp->l << ")\n";
//              cout << "todie at:" << todie << ". a at:" << a << "\n";
//              cout << "  tmp at:" << tmp << "\n";
//              todie->edgeList->DelEdgeToVertex((void*)tmp);
//              cout << "On vertex (" << tmp->t << "," << tmp->w << ",";
//              cout << tmp->l << "):Del edge to (" << todie->t << "," << todie->w << ",";
//              cout << todie->l << ")\n";
//              tmp->edgeList->DelEdgeToVertex((void*)todie);
                }
        delete todie->edgeList;
        RemoveFromSearchList(todie);

        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
                {
                vertexHead = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
                tmp = (*todie).prev;
                (*tmp).next = (*todie).next;
                tmp = (*todie).next;
                (*tmp).prev = (*todie).prev;
                free(todie);
                return;
                }
        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
                vertexHead = tmp = (*todie).next;
                (*tmp).prev = (VERTEX_NODE*)NULL;
                free(todie);
```

```
		return;
		}
	if ( ((*todie).prev != (VERTEX_NODE*)NULL)
	&& ((*todie).next == (VERTEX_NODE*)NULL) )
		{
		tmp = (*todie).prev;
		(*tmp).next = (VERTEX_NODE*)NULL;
		free(todie);
		return;
		}
	}


int VertexList::FindPath()
   {
	EDGE_NODE *e;
	int goalFound=FALSE;
	float dist;
	VERTEX_NODE *cur, *adj;

	if (searchHead->nodeType == START)
		searchMarker = searchHead->searchMarker;
	else
		return(FALSE);
	for (cur = searchHead; cur != (VERTEX_NODE*)NULL; cur = searchHead)
		{
		if ((cur->searchDist >= searchTrimDist)
		&& (searchTrimDist > 0.0))
			{
			RemoveFromSearchList(cur);
			continue;
			}
		for(e = cur->edgeList->GetFirstEdge();
			e != (EDGE_NODE*)NULL;
			e = cur->edgeList->GetNextEdge(e))
			{
	  adj = FindVertex(e->t, e->w, e->l);
	  if (adj == (VERTEX_NODE*)NULL)
	     continue;

//	    adj = (VERTEX_NODE*) e->otherVertex;
			dist = cur->searchDist + e->dist;
			if ( (adj->searchMarker != searchMarker)
			|| ((adj->searchMarker == searchMarker)
				&& (adj->searchDist > dist)) )
				{
				if ((dist < searchTrimDist)
				|| (searchTrimDist <= 0.0))
					{
					adj->pathFrom = cur;
					adj->searchMarker = searchMarker;
					adj->searchDist = dist;
					AddToSearchList(adj);
					if (adj->nodeType == GOAL)
						{
						goalFound = TRUE;
						searchTrimDist = dist;
						MarkPath(adj);
						TrimSearchList();
						}
					}
				}
			}
		RemoveFromSearchList(cur);
		}
	return(goalFound);
	}


VERTEX_NODE* VertexList::FindVertex(int t, int w, int l)
   {
   VERTEX_NODE* cur;

   for(cur = GetFirstVertex();
      cur != (VERTEX_NODE*)NULL;
      cur = GetNextVertex(cur))
      {
      if ((cur->t == t)
      && (cur->w == w)
      && (cur->l == l))
         return(cur);		// found it

      if ((cur->t >= t)
```

```
        && (cur->w >= w)
        && (cur->l >= l))
           break;              // passed it - it's not in the list
        }
    return((VERTEX_NODE*)NULL);
    }


VERTEX_NODE* VertexList::GetFirstVertex()
    {
    return(vertexHead);
    }


VERTEX_NODE* VertexList::GetNextVertex(VERTEX_NODE*cur)
    {
    return(cur->next);
    }


VERTEX_NODE* VertexList::GetStartVertex()
    {
    VERTEX_NODE*tmp;

    for(tmp = GetFirstVertex();
        tmp != (VERTEX_NODE*)NULL;
        tmp = GetNextVertex(tmp))
        {
                if (tmp->nodeType == START)
           break;
        }
    return(tmp);
    }


void VertexList::InsertNewVertex(VERTEX_NODE*a)
        {
        VERTEX_NODE *cur;

        if (a == (VERTEX_NODE*)NULL)
                return;

        if (vertexHead == (VERTEX_NODE*)NULL)
                {
                vertexHead = a;
                (*a).prev = (*a).next = (VERTEX_NODE*)NULL;
                return;
                }

        for(cur=vertexHead; cur != (VERTEX_NODE*)NULL;   cur=(*cur).next)
                {
                if ( (a->t > cur->t)
      || ((a->t == cur->t)
         && (a->w > cur->w))
       || ((a->t == cur->t)
         && (a->w == cur->w)
              && (a->l > cur->l)) )
                        {                               // insert after cur
                        if ((*cur).next == (VERTEX_NODE*)NULL)
                                {
                                (*a).next = (*cur).next;    // eol - append new node
                                (*a).prev = cur;
                                (*cur).next = a;
                                break;
                                }
                        else
                                continue;                   // get next node
                        }
                if ( (a->t < cur->t)
      || ((a->t == cur->t)
         && (a->w < cur->w))
      || ((a->t == cur->t)
                  && (a->w == cur->w)
                        && (a->l < cur->l)) )
                        {                               // insert before cur
                        if (cur->prev == (VERTEX_NODE*)NULL)
                                {
                                a->prev = (VERTEX_NODE*)NULL;// at start of list
                                a->next = cur;
                                cur->prev = a;
                                vertexHead = a;
                                break;
                                }
                        else
```

```
					{
					a->prev = cur->prev;		// in middle/end of list
					a->next = cur;
					cur->prev = a;
					cur = a->prev;
					cur->next = a;
					break;
					}
				}
			if ((a->t == cur->t)
		&& (a->w == cur->w)
			&& (a->l == cur->l))
				{				// insert after cur at eol
				if ((*cur).next == (VERTEX_NODE*)NULL)
					{
					(*a).next = (VERTEX_NODE*)NULL;
					(*a).prev = cur;
					(*cur).next = a;
					break;
					}
				else
					continue;
				}
			}
//	cout << "Inserted vertex (";
// cout << (*a).t << "," << (*a).w << "," << (*a).l << ")\n";
	}


void VertexList::ListAllVertices()
	{
	VERTEX_NODE *cur=vertexHead;

   while (cur != (VERTEX_NODE*)NULL)
	{
		cout << "Vertex:" <<  (*cur).nodeType << ": at (";
		cout << (*cur).t << "," << (*cur).w << "," << (*cur).l << ")\n";
		cur->edgeList->ListAllEdges();
	cur = (*cur).next;
	}
   }


void VertexList::ListSearchList()
	{
	VERTEX_NODE*cur;

	if (searchHead == (VERTEX_NODE*)NULL)
		{
		cout << "Empty SearchList! (marker=" << searchMarker << ")\n";
		return;
		}

	cout << "List of nodes in Search list (marker=" << searchMarker << ")\n";
	cur = searchHead;
	do
		{
		cout << "SearchList node (" << cur->t << "," << cur->w << "," << cur->l << ") ";
	cout << "Dist = " << cur->searchDist << "\n";
		cur=cur->searchNext;
		}
	while (cur != searchHead);
	cout << "End of SearchList\n";
	}


void VertexList::MarkPath(VERTEX_NODE*v)
	{
	VERTEX_NODE*tmp;

	if (v == (VERTEX_NODE*)NULL)
		return;

	cout << "Path from Goal to Start\n";
	v->pathTo = (VERTEX_NODE*)NULL;
	do
		{
		tmp = v->pathFrom;
//		cout << "(" << v->t << "," << v->w << "," << v->l << ")\n";
		if (tmp != (VERTEX_NODE*)NULL)
			{
			tmp->pathTo = v;
			v = v->pathFrom;
			}
```

```
		}
	while (tmp != (VERTEX_NODE*)NULL);
	}


void VertexList::RemoveFromSearchList(VERTEX_NODE*a)
	{
	VERTEX_NODE *cur;

	if ((a->searchPrev == (VERTEX_NODE*)NULL)
	|| (a->searchNext == (VERTEX_NODE*)NULL))
	{
	if (searchHead == a)
	   searchHead = (VERTEX_NODE*)NULL;
	return;
	}

	cur=a->searchPrev;
	if (cur == a)
		{
	searchHead = (VERTEX_NODE*)NULL;
//		cout << "Removed Search Node (";
//	cout << a->t << "," << a->w << "," << a->l << "). SearchList empty\n";
		}
	else
		{
		cur->searchNext = a->searchNext;
		cur = a->searchNext;
		cur->searchPrev = a->searchPrev;
		if (searchHead == a)
			searchHead = a->searchNext;
//		cout << "Removed Search Node (";
//	cout << a->t << "," << a->w << "," << a->l << ").\n";
		}
	a->searchPrev = a->searchNext = (VERTEX_NODE*)NULL;
	}


void VertexList::TrimSearchList()
   {
	VERTEX_NODE *cur, *tmpPtr=(VERTEX_NODE*)0;
	int trimmed=TRUE;

//	cout << "Trimming search list to " << searchTrimDist << " or less\n";
	if ((searchHead == (VERTEX_NODE*)NULL)
   || (searchTrimDist < 0.0))
		return;

   cur = searchHead;
	while ((tmpPtr != cur) || (trimmed == TRUE))
		{
		if (trimmed==TRUE)
			{
			tmpPtr=cur;
			trimmed=FALSE;
			}
		if ((*cur).searchDist >= searchTrimDist)
			{
//			cout << "Trimmed out (";
//		cout << cur->t << "," << cur->w << "," << cur->l << ") ";
//		cout << "Dist was " << cur->searchDist << "\n";
			if (cur == cur->searchNext)
				{
		RemoveFromSearchList(cur);
				break;
				}
			else
				{
				tmpPtr = cur->searchNext;
				RemoveFromSearchList(cur);
				cur = tmpPtr;
				trimmed = TRUE;
				}
			}
		else
			cur = cur->searchNext;
		}
//   cout << "-------\n";
   }
```

B.5. Standard Priority First Graph Theory

The Standard Priority First Search Algorithm for Graph Theory which was implemented as part of this project was coded in ANSI C++. The filenames for each of the separate source code files were supplied inside C++ format comments (i.e. //) at the beginning of each file listing. A detailed explanation of the design behind this program was presented in Chapter 4.

```
// main.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <sys\timeb.h>
#include <dir.h>
#include <ctype.h>
#include "edge.hpp"
#include "vertex.hpp"
#include "object.hpp"
#include "domain.hpp"
#include "bench.hpp"

#define MAPFILE_MASK "MAP*.DAT"

int main(int, char**);
void BuildVertexListFromDomain(VertexList&, Domain&);
int CalcRobotDir(VERTEX_NODE*);
void FindAllEdges(VertexList&, Domain&);
void MemStatus(char*);
int SetWorkingDir(void);
void RenameMapFile(char*);


int main(int argc, char** argv)
	{
	char* shortName;
	struct ffblk mapfile;

	for(;;)
		{
		shortName = strstr(argv[0], "\\");
		if (shortName == (char*)NULL)
			{
			shortName = argv[0];
			cout << "This is " << shortName << "\n";
			break;
			}
		else
			argv[0] = shortName+1;
		}

	MemStatus("Free memory before mainloop in main: ");

	if (!SetWorkingDir())
		{
		cout << "Program exiting gracefully\n";
		return(1);
		}
	if (findfirst(MAPFILE_MASK, &mapfile, 0))
		{
		cout << "Program exiting gracefully - no map files found\n";
		return(1);
		}
	do
		{
		char tmp[256];
		int i;
		Benchmark stopWatch(shortName);

		sprintf(tmp, "Starting on %s: ", mapfile.ff_name);
		MemStatus(tmp);
		for (i=0; i<NUMTIMES; i++)
			{
			int flag, timeTaken;
			float distTravelled;
			Domain theWorld(10, 20, 20, mapfile.ff_name);
```

```
		timeTaken = 0;
		distTravelled = 0.0;
		stopWatch.IterStart(i, mapfile.ff_name);
		stopWatch.Click();
		for(;;)
			{
			VertexList vertList;

			BuildVertexListFromDomain(vertList, theWorld);
			FindAllEdges(vertList, theWorld);
			stopWatch.Click();
			theWorld.DrawDomain();
			stopWatch.Click();
			flag = vertList.FindPath();
			stopWatch.Click();
			if (flag == TRUE)
				cout << "Path found the GOAL!\n";
			else
				cout << "No path found to GOAL!\n";

			if (flag)
				{
				VERTEX_NODE*t1, *t2;

				t1 = vertList.GetStartVertex();
				if (t1 != (VERTEX_NODE*)NULL)
					{
					t2 = t1->pathTo;
					if (t2 != (VERTEX_NODE*)NULL)
						{
						if ((t1->w == t2->w)
						&& (t1->l == t2->l)
						&& (t2->nodeType == GOAL))
							{
							cout << "MADE IT TO THE GOAL!\n";
							break;
							}
						distTravelled += theWorld.MoveRobot(t1->t, t1->w, t1->l,

		 t2->t, t2->w, t2->l);
						}
					}
				}
			theWorld.AdvanceTime();
			timeTaken++;
			}
		stopWatch.IterStop(timeTaken, distTravelled);
		}
	stopWatch.LogCalcs();
	RenameMapFile(mapfile.ff_name);
//	break;
	}
	while (!findnext(&mapfile));

	MemStatus("Free memory after mainloop in main: ");
	return(0);
	}


void BuildVertexListFromDomain(VertexList &vList, Domain &domain)
	{
	char type;
	int t=domain.GetDomainTimeSlices(), a;
	int w=domain.GetDomainWidth(), b;
	int l=domain.GetDomainLength(), c;
	VERTEX_NODE*newPtr, *gPtr, *sPtr;

	for(gPtr = sPtr = (VERTEX_NODE*)NULL, a=0; a<t; a++)
	  {
		for(b=0; b<w; b++)
			{
			for(c=0; c<l; c++)
				{
				type = domain.GetPointType(a, b, c);
				if ((type == VERTEX)
				|| (type == START)
				|| (type == GOAL))
					{
					newPtr = vList.BuildNewVertex(a, b, c, type);
					vList.InsertNewVertex(newPtr);
					}
	    if (type == GOAL)
	       gPtr = newPtr;
	    if (type == START)
```

```
                sPtr = newPtr;
                        }
                    }
                }
    if ((gPtr != (VERTEX_NODE*)NULL)
    && (sPtr != (VERTEX_NODE*)NULL))
        {
        sPtr->searchDist = 0.0;
                vList.AddToSearchList(sPtr);
        }
        }


void FindAllEdges(VertexList &vList, Domain&domain)
        {
        EDGE_NODE*e;
        VERTEX_NODE*a,*b;
        float dist;

        for(a = vList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = vList.GetNextVertex(a))
                {
                for(b = vList.GetNextVertex(a);
                        b != (VERTEX_NODE*)NULL;
                        b = vList.GetNextVertex(b))
          {
          if (a == b)
                                continue;

                        if ((a->t == b->t)
                        && (a->w == b->w)
                        && (a->l == b->l))
                                continue;

          dist = domain.CheckLine(a->t, a->w, a->l, b->t, b->w, b->l);
          if (dist > 0.0)
              {
                                e = a->edgeList->BuildNewEdge(b->t, b->w, b->l, dist);
                                a->edgeList->InsertNewEdge(e);
                                }
                        dist = domain.CheckLine(b->t, b->w, b->l, a->t, a->w, a->l);
                        if (dist > 0.0)
                                {
                                e = b->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                                b->edgeList->InsertNewEdge(e);
              }
          }
                }
        }


void MemStatus(char *StatusMessage)
        {
        char tmp[256];
        fstream debugFile;
        long MemLeft;

        debugFile.open("DEBUG.LOG", ios::app);
        if (!debugFile)
                cout << "Unable to open DEBUG.LOG\n";

        MemLeft = (long) coreleft();
        sprintf(tmp, "%s%ld\n", StatusMessage, MemLeft);
        debugFile.write(tmp, strlen(tmp));
        debugFile.close();
//      cout << StatusMessage << MemLeft << "\n";
        }


int SetWorkingDir()
        {
        char mapdir[256];

        cout <<"Enter the directory containing map files, or \"q\" for quit:";
        cin >> mapdir;
        if (toupper(mapdir[0]) == 'Q')
                {
                cout << "Quitting...\n";
                return(FALSE);
                }
        if (chdir(mapdir))
                {
                cout << "The directory " << mapdir << " could not be found.\n";
```

```
                return(FALSE);
                }
        cout << "Made " << mapdir << " the current directory.\n";
        return(TRUE);
        }


void RenameMapFile(char*filename)
        {
        char newfilename[128];
        char* ch;
        int i=(int)'.';

        strcpy(newfilename, filename);
        ch = strrchr(newfilename,i);
        if (ch != (char*)NULL)
                {
                strcpy(ch, ".bak");
                rename(filename, newfilename);
                }
        else
                exit(1);
        }

// bench.hpp
#define FALSE 0
#define TRUE !FALSE
#define NUMTIMES 10
#define MAXFILENAME 13

class Benchmark
        {
        public:
                Benchmark(char*);
                ~Benchmark();
                void Click(void);
                void IterStart(int, char*);
                void IterStop(int, float);
                void LogCalcs(void);
        private:
                void Diff(struct timeb*, struct timeb*, struct timeb*);

                fstream logFile;
                fstream avgFile;
                char mapFileName[MAXFILENAME];
                struct timeb benchmarks[NUMTIMES][2];
                float distRobotTravelled[NUMTIMES];
                int timeTaken[NUMTIMES];
                int currentIter;
                int clickToggleFlag;
                struct timeb elapsedTime, computeTime;
                struct timeb startTime, clickOnTime, clickOffTime;
        };

// bench.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <sys\timeb.h>
#include "bench.hpp"


Benchmark::Benchmark(char *fileName)
        {
        char *tmp;
        char logname[MAXFILENAME];

        while((tmp = strchr(fileName, '\\')) != (char*)NULL)
                fileName = tmp+1;
        strcpy(logname, fileName);
        if ((tmp = strchr(logname, '.')) != (char*)NULL)
                strcpy(tmp, ".LOG");
        else
                strcat(logname, ".LOG");
        logFile.open(logname, ios::app);
        if (!logFile)
                cout << "Unable to open " << logname << "\n";

        if ((tmp = strchr(logname, '.')) != (char*)NULL)
                strcpy(tmp, ".AVG");
        else
```

```
		strcat(logname, ".AVG");
	avgFile.open(logname, ios::app);
	if (!avgFile)
		cout << "Unable to open " << logname << "\n";

	cout << "Initialised Benchmark class\n";
	}


Benchmark::~Benchmark()
	{
	if (logFile)
		{
		logFile.flush();
		logFile.close();
		}
	if (avgFile)
		{
		avgFile.flush();
		avgFile.close();
		}
	cout << "Closed log files and Destroying Benchmark class\n";
	}


void Benchmark::Click()
	{
	switch(clickToggleFlag)
		{
		case TRUE:
			ftime(&clickOnTime);
			clickToggleFlag = FALSE;
			break;
		case FALSE:
		default:
			ftime(&clickOffTime);
			Diff(&clickOnTime, &clickOffTime, &computeTime);
			clickToggleFlag = TRUE;
		}
	}


void Benchmark::Diff(struct timeb*start, struct timeb*stop, struct timeb*diff)
	{
	if ((*stop).millitm < (*start).millitm)
		{
		(*stop).millitm += (short)1000; /* carry when subtracting, stops*/
		(*start).time += 1L;            /* negative wraparound problems!*/
		}
	(*diff).millitm += (*stop).millitm - (*start).millitm;
	(*diff).time += (long)((*diff).millitm / (short)1000);
	(*diff).millitm %= (short)1000;
	(*diff).time += ((*stop).time-(*start).time);
	}


void Benchmark::IterStart(int i, char*s)
	{
	currentIter = i;
	clickToggleFlag = TRUE;
	computeTime.time = elapsedTime.time = 0L;
	computeTime.millitm = elapsedTime.millitm = 0;

	strcpy(mapFileName, s);

	ftime(&startTime);
	}


void Benchmark::IterStop(int t, float distTravelled)
	{
	struct timeb stopTime;

	ftime(&stopTime);
	Diff(&startTime, &stopTime, &elapsedTime);

	benchmarks[currentIter][0].time = elapsedTime.time;
	benchmarks[currentIter][0].millitm = elapsedTime.millitm;
	benchmarks[currentIter][1].time = computeTime.time;
	benchmarks[currentIter][1].millitm = computeTime.millitm;
	timeTaken[currentIter] = t;
	distRobotTravelled[currentIter] = distTravelled;
	}
```

```
void Benchmark::LogCalcs()
	{
	char tmp[256];
	float avgDist;
	struct timeb avg;
	int i, avgTimeTaken;

	for(i=0, avg.time=0L, avg.millitm=0, avgDist=0.0, avgTimeTaken=0;
		i<NUMTIMES;
		i++)
		{
		sprintf(tmp, "%s (%02d) Elapsed time:%05ld.%03d\n", mapFileName, i,
					benchmarks[i][0].time, benchmarks[i][0].millitm);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Compute time:%05ld.%03d\n", mapFileName, i,
					benchmarks[i][1].time, benchmarks[i][1].millitm);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Dist travelled:%f\n", mapFileName, i,
					distRobotTravelled[i]);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Time Slices Taken:%d\n", mapFileName, i,
					timeTaken[i]);
		logFile.write(tmp, strlen(tmp));
		avg.time+=benchmarks[i][0].time;
		avg.millitm+=benchmarks[i][0].millitm;
		if (avg.millitm % 1000 != avg.millitm)
			{
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;
			}
		}

	sprintf(tmp, "%s Tot. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));
	cout << avg.time << "." << avg.millitm << "\n";
	cout << tmp << "\n";

	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;

	sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	avgFile.write(tmp, strlen(tmp));
//	sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));

	for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
		{
		avg.time+=benchmarks[i][1].time;
		avg.millitm+=benchmarks[i][1].millitm;
		if (avg.millitm % 1000 != avg.millitm)
			{
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;
			}
		}
	sprintf(tmp, "%s Tot. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));
	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;
	sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	avgFile.write(tmp, strlen(tmp));
//	sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));

	for(i=0, avgTimeTaken=0; i<NUMTIMES; i++)
		avgTimeTaken+=timeTaken[i];
	avgTimeTaken = avgTimeTaken / NUMTIMES;
	sprintf(tmp, "%s Avg. Time Taken:%d\n", mapFileName, avgTimeTaken);
	avgFile.write(tmp, strlen(tmp));
	logFile.write(tmp, strlen(tmp));

	for(i=0, avgDist=0.0; i<NUMTIMES; i++)
		avgDist += distRobotTravelled[i];
	avgDist = avgDist / ((float)NUMTIMES);
	sprintf(tmp, "%s Avg. Dist Travelled:%f\n", mapFileName, avgDist);
	avgFile.write(tmp, strlen(tmp));
	logFile.write(tmp, strlen(tmp));
```

```
        sprintf(tmp, "%s ====================\n", mapFileName);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s ====================\n", mapFileName);
        logFile.write(tmp, strlen(tmp));
        }

// domain.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00

#define FROM_FRONT 0
#define FROM_BACK 1
#define FROM_LEFT 2
#define FROM_RIGHT 3
#define FROM_FRONTRIGHT 4
#define FROM_FRONTLEFT 5
#define FROM_BACKRIGHT 6
#define FROM_BACKLEFT 7
#define FROM_UP 8
#define FROM_DOWN 9
#define FROM_NOWHERE 10

#define NOCOST (float)0.0
#define NORMAL (float)1.0
#define NORMAL_DIAG (float)1.414214
#define BLOCKED (float)1000.0

typedef struct
        {
        float dist;
        int from;
        char type;
        float cost[9];
        } POINT;

class Domain
        {
        public:
                Domain(int, int, int, char*);
                ~Domain();
                void AdvanceTime(void);
        float CheckLine(int, int, int, int, int, int);
                void DrawDomain(void);
        int GetDomainLength(void) { return(domainLength); }
        int GetDomainTimeSlices(void) { return(domainTimeSlices); }
        int GetDomainWidth(void) { return(domainWidth); }
                POINT* GetPoint(int, int, int);
        float GetPointCost(int, int, int, int);
                char GetPointFrom(int, int, int);
                char GetPointType(int, int, int);
                int IsPointClear(int, int, int);
                int IsPointGoal(int, int, int);
                int IsPointNearObject(int, int, int);
                int IsPointObject(int, int, int);
                int IsPointStart(int, int, int);
        int IsPointVertex(int, int, int);
                float MoveRobot(int, int, int, int, int, int);
                void SetPointFrom(int, int, int, int);
                void SetPointType(int, int, int, char);
        private:
                void AgeTimeSlices(void);
        int CalcRobotDir(int, int, int, int, int, int);
                int ClearAdjPointOK(int, int, int);
                void ClearMobileObject(int, int, int);
                void ClearVerticesInTimeSlice(int);
                void DrawTimeSlice(int);
                void InitTimeSlice(int);
                void MarkMobileObject(int, int, int);
                void MoveMobileObject(OBJECT_NODE*);
```

```
            void MoveMobileObjects(void);
            void SetAdjObjsInTimeSlice(int);
            void SetGoalFromFile(char*);
            void SetMobileObjsFromFile(char*);
            void SetPermObjsInTimeSlice(int);
            void SetStartFromFile(char*);
            void SetVerticesInTimeSlice(int);

            ObjectList objList;
            POINT** domainHead;
        int domainWidth;
        int domainLength;
        int domainTimeSlices;
        };

// domain.cpp
#include <iostream.h>
#include <fstream.h>
#include <alloc.h>          // for coreleft()
#include <stdio.h>
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <conio.h>
#include "object.hpp"
#include "domain.hpp"

Domain::Domain(int numTimeSlices, int width, int length, char*mapFileName)
        {
        int i;

        cout << "Contructing Domain class\n";
        domainTimeSlices = numTimeSlices;
        domainWidth = width;
        domainLength = length;

        cout << "FreeHeap:" << farcoreleft() << "\n";
        cout << "Amount needed for one timeslice:";
        cout << width*length*sizeof(POINT) << "\n";

        domainHead = (POINT**)farcalloc(numTimeSlices, sizeof(POINT**));
        if (domainHead == (POINT**)NULL)
                {
                cout << "Not enough memory to build model of world\n";
                domainHead = (POINT***)NULL;
                domainTimeSlices = domainWidth = domainLength = 0;
                return;
                }

        for(i=0; i<numTimeSlices; i++)
                {
                domainHead[i] = (POINT*)farcalloc(width*length, sizeof(POINT));
                cout << "FreeHeap after timeslice allocated:" << farcoreleft() << "\n";
                if (domainHead[i] == (POINT*)NULL)
                        {
                        if (i==0)
                                cout << "Not enough memory to build model of world\n";
                        else
                                cout << "Only enough memory to build " << i << " of the ";
                                cout << numTimeSlices << " timeslices in model of world\n";
                        domainTimeSlices = i;
                        break;
                        }
                InitTimeSlice(i);
                SetPermObjsInTimeSlice(i);
                }
        SetStartFromFile(mapFileName);
        SetGoalFromFile(mapFileName);
        SetMobileObjsFromFile(mapFileName);
        for(i=0; i<domainTimeSlices; i++)
                {
                SetAdjObjsInTimeSlice(i);
                SetVerticesInTimeSlice(i);
                }
        for(i=0; i<domainTimeSlices-1; i++)
                AdvanceTime();
        cout << "FreeHeap after Domain allocated:" << farcoreleft() << "\n";
        }



Domain::~Domain()
        {
        POINT *tmp;
        int i;
```

```
        cout << "Destructing Domain class\n";
        if (domainHead == (POINT**)NULL)
                {
                cout << "Not freeing domain - DomainHead was NULL\n";
                return;
                }
        for(i=0; i<domainTimeSlices; i++)
                {
                tmp = domainHead[i];
                farfree(tmp);
                }
        farfree(domainHead);
        domainHead = (POINT**)NULL;
        cout << "FreeHeap after Domain deallocated:" << farcoreleft() << "\n";
//      getch();
        }


void Domain::AdvanceTime()
        {
        AgeTimeSlices();
//      DrawDomain();
        ClearVerticesInTimeSlice(domainTimeSlices-1);
        MoveMobileObjects();
//      DrawDomain();
        SetVerticesInTimeSlice(domainTimeSlices-1);
//      DrawDomain();
        }


///////////////////////////////////////
//
// Copy the contents of every timeslice
// into the previous timeslice
// [0] = [1], [1] = [2], etc.
// Leave the last timeslice unchanged.
// Another routine will decide the moves
// for all the mobile objects.
//
// A quick way to do this is moving ptrs
// to the timeslices and only copying the
// contents of the last timeslice over the
// contents of the first timeslice
//
///////////////////////////////////////
void Domain::AgeTimeSlices()
        {
        int i, j;
        POINT *tmp;

        tmp = domainHead[0];
        for(i=0, j=1; i<domainTimeSlices-1; i++, j++)
                domainHead[i] = domainHead[j];
        domainHead[domainTimeSlices-1] = tmp;
        _fmemcpy(domainHead[domainTimeSlices-1],
                                domainHead[domainTimeSlices-2],
                                sizeof(*domainHead[domainTimeSlices-1]));
        }


int Domain::CalcRobotDir(int st, int sw, int sl, int et, int ew, int el)
   {
        if ((!IsPointStart(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(clear);

        if ((ew - sw > 0)
        && (el - sl > 0))
                {
                sw++, sl++;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
                || (IsPointGoal(st, sw, sl)))
                        return(frontright);
                else
                        return(clear);
                }

        if ((ew - sw > 0)
        && (el - sl == 0))
                {
                sw++;
                if ((IsPointClear(st, sw, sl))
```

```
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(right);
        else
                return(clear);
        }

if ((ew - sw > 0)
&& (el - sl < 0))
        {
        sw++, sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(backright);
        else
                return(clear);
        }

if ((ew - sw == 0)
&& (el - sl > 0))
        {
        sl++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(front);
        else
                return(clear);
        }

if ((ew - sw == 0)  // silly, but just in case
&& (el - sl == 0))
        {
        return(clear);
        }

if ((ew - sw == 0)
&& (el - sl < 0))
        {
        sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(back);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl > 0))
        {
        sw--, sl++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(frontleft);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl == 0))
        {
        sw--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(left);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl < 0))
        {
        sw--, sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(backleft);
        else
                return(clear);
        }
```

```
	return(clear);
	}


float Domain::CheckLine(int st, int sw, int sl, int et, int ew, int el)
	{
//	static int displayCounter=1;
	int tmpt, tmpw, tmpl;
	float dist;

	for(dist=0.0, tmpt=et-st, tmpw=ew-sw, tmpl=el-sl;
		(tmpt != 0) || (tmpw != 0) || (tmpl != 0);
		tmpt=et-st, tmpw=ew-sw, tmpl=el-sl)
		{
		if (tmpt < 0)
			{
			return(-1.0);
			}
		if ((tmpw > 0)
		&& (tmpl > 0))
			{
			dist += GetPointCost(st, sw++, sl++, frontright);
			if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
	if ((tmpw > 0)
		&& (tmpl == 0))
			{
	  dist += GetPointCost(st, sw++, sl, right);
		if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
			}
		if ((tmpw > 0)
		&& (tmpl < 0))
			{
	  dist += GetPointCost(st, sw++, sl--, backright);
			if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
		if ((tmpw < 0)
		&& (tmpl > 0))
			{
	  dist += GetPointCost(st, sw--, sl++, frontleft);
		if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
		if ((tmpw < 0)
		&& (tmpl == 0))
			{
	  dist += GetPointCost(st, sw--, sl, left);
		if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
		if ((tmpw < 0)
		&& (tmpl < 0))
			{
	  dist += GetPointCost(st, sw--, sl--, backleft);
			if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
			}
		if ((tmpw == 0)
		&& (tmpl > 0))
			{
	  dist += GetPointCost(st, sw, sl++, front);
		if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
		if ((tmpw == 0)
		&& (tmpl < 0))
```

```
                    {
        dist += GetPointCost(st, sw, sl--, back);
             if ((!IsPointClear(st, sw, sl))
             && ((st != et) || (sw != ew) || (sl != el)))
                    return(-1.0);
        continue;
        }
      if ((tmpw == 0)
      && (tmpl == 0)
      && (tmpt >0))
                    {
        dist += GetPointCost(st++, sw, sl, up);
             if ((!IsPointClear(st, sw, sl))
             && ((st != et) || (sw != ew) || (sl != el)))
                    return(-1.0);
        continue;
        }
             }
//     cout << "CheckLine:" << displayCounter++ << ":returned" << dist << "\n";
       return(dist);
       }


int Domain::ClearAdjPointOK(int t, int w, int l)
       {
       char type;
       int a,b;

       for(a=w-1;a<=w+1;a++)
              {
              if ((a < 0) || (a >= domainWidth))
                     continue;
              for(b=l-1;b<=l+1;b++)
                     {
                     if ((b < 0) || (b >= domainLength))
                            continue;
                     if ((a == w) && (b == l))
                            continue;

         type = GetPointType(t, a, b);
                     if ((type == OBJECT)
                     ||  (type == MOBILE_OBJECT))
                            return(FALSE);
                     }
              }
       return(TRUE);
       }


void Domain::ClearMobileObject(int t, int w, int l)
   {
       int i, j;

       if (GetPointType(t, w, l) == MOBILE_OBJECT)
              {
              if (ClearAdjPointOK(t, w, l))
                     {
                     SetPointType(t, w, l, CLEAR);
                     SetPointFrom(t, w, l, FROM_NOWHERE);
                     }
              else
                     {
                     SetPointType(t, w, l, ADJ_TO_OBJECT);
                     SetPointFrom(t, w, l, FROM_NOWHERE);
                     }
              }
       for(i=w-1, j=l-1; j<l+2;)
              {
      if ((GetPointType(t, i, j) == ADJ_TO_OBJECT)
              && (ClearAdjPointOK(t, i, j)))
              {
              SetPointType(t, i, j, CLEAR);
                     SetPointFrom(t, i, j, FROM_NOWHERE);
                     }
              if (i == w+1)
                     {
                     i = w-1;
                     j++;
                     }
              else
                     i++;
              }
       }
```

```
void Domain::ClearVerticesInTimeSlice(int t)
	{
	int w, l;

	for(w=0; w<domainWidth; w++)
		{
		for(l=0; l<domainLength; l++)
			{
			if (GetPointType(t, w, l) == VERTEX)
		{
				SetPointFrom(t, w, l, FROM_NOWHERE);
				SetPointType(t, w, l, CLEAR);
		continue;
		}
			if (GetPointType(t, w, l) == MOBILE_OBJECT)
				{
		ClearMobileObject(t, w, l);
				}
			}
		}
	}


void Domain::DrawDomain()
	{
	int i;

	for (i=0; i< domainTimeSlices; i++)
		{
		DrawTimeSlice(i);
//		getch();
		}
	}


void Domain::DrawTimeSlice(int timeSlice)
	{
	char type;
	int a,b,y;

	clrscr();
	gotoxy(1,1);
	cout << "time=t+" << timeSlice;

	for(b=0, y=2;b<domainLength; b++, y++)
		{
		gotoxy(1, y);
		for(a=0; a<domainWidth; a++)
			{
			type = GetPointType(timeSlice, a, b);
			switch(type)
				{
				case GOAL:
				case START:
				case ADJ_TO_OBJECT:
				case OBJECT:
				case MOBILE_OBJECT:
				case VERTEX:
					cout << type;
					break;
				case CLEAR:
					switch(GetPointFrom(timeSlice, a, b))
						{
						case FROM_RIGHT:
							cout << "R";
							break;
						case FROM_LEFT:
							cout << "L";
							break;
						case FROM_UP:
							cout << "U";
							break;
						case FROM_DOWN:
							cout << "D";
							break;
						case FROM_FRONT:
							cout << "F";
							break;
						case FROM_BACK:
							cout << "B";
							break;
						case FROM_BACKLEFT:
							cout << "T";
```

```
                                                break;
                                        case FROM_BACKRIGHT:
                                                cout << "U";
                                                break;
                                        case FROM_FRONTLEFT:
                                                cout << "V";
                                                break;
                                        case FROM_FRONTRIGHT:
                                                cout << "W";
                                                break;
                                        case FROM_NOWHERE:
                                        default:
                                                cout << ".";
                                                break;
                                        }
                                break;
                        default:
                                cout << "?";
                                break;
                        }
                }
        cout << "\n";
        }
}


POINT* Domain::GetPoint(int timeSlice, int width, int length)
        {
   POINT *tmp, *tmp2;

   tmp = domainHead[timeSlice];
        tmp2 = tmp + (domainWidth*width) + length;
   return(tmp2);
   }


float Domain::GetPointCost(int timeSlice, int width, int length, int dir)
   {
   POINT *tmp;

   tmp = GetPoint(timeSlice, width, length);
        return(tmp->cost[dir]);
   }

char Domain::GetPointFrom(int timeSlice, int width, int length)
   {
   POINT *tmp;

   tmp = GetPoint(timeSlice, width, length);
        return(tmp->from);
   }


char Domain::GetPointType(int timeSlice, int width, int length)
        {
   POINT *tmp;

        tmp = GetPoint(timeSlice, width, length);
        return(tmp->type);
        }


void Domain::InitTimeSlice(int timeSlice)
        {
        POINT*tmp;
   int a,b;

   for(a=0;a<domainWidth;a++)
                {
                for(b=0;b<domainLength;b++)
                        {
        tmp = GetPoint(timeSlice, a, b);
                        tmp->dist=0.0;
                        tmp->from=FROM_NOWHERE;
                        tmp->type=CLEAR;

                        if (timeSlice == domainTimeSlices-1)
                                tmp->cost[FROM_UP] = BLOCKED;
                        else
                                tmp->cost[FROM_UP] = NORMAL;

                        if ((a == 0) && (b == 0))
                                {
                                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
```

```
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
        tmp->cost[FROM_BACK] = BLOCKED;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = BLOCKED;
        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a == 0) && (b < domainLength-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = BLOCKED;
        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a == 0) && (b==domainLength-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = BLOCKED;
        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
        tmp->cost[FROM_FRONT] = BLOCKED;
        continue;
        }
if ((a < domainWidth-1) && (b == 0))
        {
        tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
        tmp->cost[FROM_BACK] = BLOCKED;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a < domainLength-1) && (b < domainWidth-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a < domainWidth-1) && (b == domainLength-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
        tmp->cost[FROM_FRONT] = BLOCKED;
        continue;
        }
if ((a == domainWidth-1) && (b == 0))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
        tmp->cost[FROM_RIGHT] = BLOCKED;
        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
        tmp->cost[FROM_BACK] = BLOCKED;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a == domainWidth-1) && (b < domainWidth-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
```

```
        if (i == w+1)
            {
            i = w-1;
            j++;
            }
        else
                i++;
            }
        }


void Domain::MoveMobileObject(OBJECT_NODE*object)
        {
        int t=domainTimeSlices-1, w=(*object).w, l=(*object).l;

        ClearMobileObject(t, w, l);
        switch((*object).direction)
                {
                case front:
                        if (IsPointObject(t, w, l+1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).l++;
                        break;
                case frontleft:
                        if (IsPointObject(t, w-1, l+1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).l++, (*object).w--;
                        break;
                case left:
                        if (IsPointObject(t, w-1, l))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w--;
                        break;
                case backleft:
                        if (IsPointObject(t, w-1, l-1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w--, (*object).l--;
                        break;
                case back:
                        if (IsPointObject(t, w, l-1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).l--;
                        break;
                case backright:
                        if (IsPointObject(t, w+1, l-1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w++, (*object).l--;
                        break;
                case right:
                        if (IsPointObject(t, w+1, l))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w++;
                        break;
                case frontright:
                        if (IsPointObject(t, w+1, l+1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w++, (*object).l++;
                        break;
                default:
                        cout << "*** unknown direction for mobile object ignored ***\n";
                        break;
                }
        MarkMobileObject(t, object->w, object->l);
        }


void Domain::MoveMobileObjects()
        {
        OBJECT_NODE*cur, *orig;

        orig = cur = objList.GetNextObject();
        if (cur == (OBJECT_NODE*)NULL)
        return;

        do
```

```
                                        tmp->cost[FROM_RIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACK] = NORMAL;
                                        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                                        tmp->cost[FROM_LEFT] = NORMAL;
                                        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                                        tmp->cost[FROM_FRONT] = NORMAL;
                                        continue;
                                        }
                                if ((a == domainLength-1) && (b == domainWidth-1))
                                        {
                                        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                                        tmp->cost[FROM_RIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACK] = NORMAL;
                                        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                                        tmp->cost[FROM_LEFT] = NORMAL;
                                        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                                        tmp->cost[FROM_FRONT] = BLOCKED;
                                        continue;
                                        }
                                }
                        }
                }


int Domain::IsPointClear(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
    || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case CLEAR:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointGoal(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
    || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case GOAL:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointNearObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
                case OBJECT:
                        return(OBJECT);
                case MOBILE_OBJECT:
                        return(MOBILE_OBJECT);
                case ADJ_TO_OBJECT:
```

```
				return(ADJ_TO_OBJECT);
			default:
				return(FALSE);
			}
	}

int Domain::IsPointObject(int t, int w, int l)
	{
	if ((w < 0)
	|| (l < 0)
	|| (t < 0)
	|| (w >= domainWidth)
	|| (l >= domainLength)
	|| (t >= domainTimeSlices))
		return(TRUE);
	switch(GetPointType(t, w, l))
		{
		case OBJECT:
			return(OBJECT);
		case MOBILE_OBJECT:
			return(MOBILE_OBJECT);
		case ADJ_TO_OBJECT:
		default:
			return(FALSE);
		}
	}


int Domain::IsPointStart(int t, int w, int l)
	{
	if ((w < 0)
	|| (l < 0)
	|| (t < 0)
	|| (w >= domainWidth)
	|| (l >= domainLength)
	|| (t >= domainTimeSlices))
		return(FALSE);
	switch(GetPointType(t, w, l))
		{
		case START:
			return(TRUE);
		default:
			return(FALSE);
		}
	}


int Domain::IsPointVertex(int t, int w, int l)
	{
	if ((w < 0)
	|| (l < 0)
	|| (t < 0)
	|| (w >= domainWidth)
	|| (l >= domainLength)
	|| (t >= domainTimeSlices))
		return(FALSE);
	switch(GetPointType(t, w, l))
		{
		case VERTEX:
			return(TRUE);
		default:
			return(FALSE);
		}
	}


void Domain::MarkMobileObject(int t, int w, int l)
	{
	int i, j;

	if ((GetPointType(t, w, l) == CLEAR)
	|| (GetPointType(t, w, l) == ADJ_TO_OBJECT))
		{
		SetPointType(t, w, l, MOBILE_OBJECT);
		SetPointFrom(t, w, l, FROM_NOWHERE);
		}
	for(i=w-1, j=l-1; j<l+2;)
		{
	if (GetPointType(t, i, j) == CLEAR)
		{
		SetPointType(t, i, j, ADJ_TO_OBJECT);
		SetPointFrom(t, i, j, FROM_NOWHERE);
		}
```

```
		{
		if ((*cur).velocity > 0)
			MoveMobileObject(cur);

	cur = objList.GetNextObject();
		} while (cur != orig);
	}


float Domain::MoveRobot(int st, int sw, int sl, int et, int ew, int el)
	{
	float dist = 0.0;
	int dir;

	if (!IsPointStart(st, sw, sl))
		return(dist);

	dir = CalcRobotDir(st, sw, sl, et, ew, el);
	SetPointType(st, sw, sl, CLEAR);
	SetPointFrom(st, sw, sl, FROM_NOWHERE);

	switch(dir)
		{
		case frontright:
			dist = GetPointCost(st, sw, sl, frontright);
			st++, sw++, sl++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw--, sl--;
				dist = 0.0;
				}
			break;
		case right:
			dist = GetPointCost(st, sw, sl, right);
			st++, sw++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw--;
				dist = 0.0;
				}
			break;
		case backright:
			dist = GetPointCost(st, sw, sl, backright);
			st++, sw++, sl--;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw--, sl++;
				dist = 0.0;
				}
			break;
		case frontleft:
			dist = GetPointCost(st, sw, sl, frontleft);
			st++, sw--, sl++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw++, sl--;
				dist = 0.0;
				}
			break;
		case left:
			dist = GetPointCost(st, sw, sl, left);
			st++, sw--;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw++;
				dist = 0.0;
				}
			break;
		case backleft:
			dist = GetPointCost(st, sw, sl, backleft);
			st++, sw--, sl--;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
```

```
                    {
                    st--, sw++, sl++;
                    dist = 0.0;
                    }
                break;
            case front:
                dist = GetPointCost(st, sw, sl, front);
                st++, sl++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                    {
                    st--, sl--;
                    dist = 0.0;
                    }
                break;
            case back:
                dist = GetPointCost(st, sw, sl, back);
                st++, sl--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                    {
                    st--, sl++;
                    dist = 0.0;
                    }
                break;
            case up:
                dist = GetPointCost(st, sw, sl, up);
                st++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                    {
                    st--;
                    dist = 0.0;
                    }
                break;
            default:
                break;
            }
        SetPointType(st, sw, sl, START);
        SetPointFrom(st, sw, sl, FROM_NOWHERE);
        return(dist);
        }


void Domain::SetAdjObjsInTimeSlice(int timeSlice)
        {
        int a,b;
        int w,l;

        for(w=0; w<domainWidth; w++)
            {
            for(l=0; l<domainLength; l++)
                {
                if (!IsPointObject(timeSlice, w, l))
                    continue;

                for(a=w-1; a<=w+1; a++)
                    {
                    if ((a < 0) || (a >= domainWidth))
                        continue;
                    for(b=l-1; b<=l+1; b++)
                        {
                        if ((b < 0) || (b >= domainLength))
                            continue;

                        if (GetPointType(timeSlice, a, b) == CLEAR)
                            {
                            SetPointFrom(timeSlice, a, b, FROM_NOWHERE);
                            SetPointType(timeSlice, a, b, ADJ_TO_OBJECT);
                            }
                        }
                    }
                }
            }
        }


void Domain::SetGoalFromFile(char*fileName)
        {
        int i, w, l;
        char recType;
```

```
        char tmp[256];
        fstream dataFile;

        dataFile.open(fileName, ios::in);
        if (!dataFile)
                cout << "Unable to open " << fileName << ". No Goal added.\n";

        for(;;)
                {
                dataFile. getline(tmp, sizeof(tmp));
                if (strlen(tmp) == 0)
                        break;
                if (tmp[0] == GOAL)
                        {
//                      cout << "The GOAL line is: " << tmp << "\n";
                        if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                                {
             for(i=0; i<domainTimeSlices; i++)
               {
                                SetPointFrom(i, w, l, FROM_NOWHERE);
                                SetPointType(i, w, l, GOAL);
               }
                                break;
                                }
                        else
                                cout << "Improperly formatted line ignored\n";
                        }
                }
        dataFile.close();
        }


void Domain::SetMobileObjsFromFile(char*fileName)
        {
        int w, l;
        char recType;
        char tmp[256];
        fstream dataFile;
        OBJECT_NODE* newObj;

        dataFile.open(fileName, ios::in);
        if (!dataFile)
                cout << "Unable to open " << fileName << ". No Mobile objs added.\n";

        for(;;)
                {
                int i;

                dataFile. getline(tmp, sizeof(tmp));
                i = strlen(tmp);
                if (i <= 0)
                        break;
                cout << "Length of input line is: " << i << "\n";
                if ((tmp[0] != GOAL)
                && (tmp[0] != START))
                        {
//                      cout << "OBJECT line is: " << tmp << "\n";
                        if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                                {
                                SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
                                SetPointType(domainTimeSlices-1, w, l, MOBILE_OBJECT);
                                newObj = objList.BuildNewObject(w, l);
                                if (newObj)
                                        {
                                        objList.InsertNewObject(newObj);
//                                      cout << "Added obj to obj list\n";
                                        }
                                }
                        else
                                cout << "Improperly formatted line ignored\n";
                        }
                }
        dataFile.close();
        }


void Domain::SetPermObjsInTimeSlice(int timeSlice)
        {
        int w, l;

        for(w=0; w<domainWidth-5; w++)
                {
                l=2;
                switch(w)
```

```
                        {
                        case 3:
                        case 4:
                        case 5:
                                break;
                        default:
                                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                                SetPointType(timeSlice, w, l, OBJECT);
//                              SetAdjObjsInTimeSlice(timeSlice, w, l);
                                break;
                        }
                }
        for(w=0; w<domainWidth-10; w++)
                {
                l=7;
                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                SetPointType(timeSlice, w, l, OBJECT);
//              SetAdjObjsInTimeSlice(timeSlice, w, l);
                }
        for(l=0, w=domainWidth-10; l<domainLength-2; l++)
                {
                switch(l)
                        {
                        case 3:
                        case 4:
                        case 5:
                        case 6:
                        case 14:
                        case 15:
                        case 16:
                                break;
                        default:
                                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                                SetPointType(timeSlice, w, l, OBJECT);
//                              SetAdjObjsInTimeSlice(timeSlice, w, l);
                                break;
                        }
                }
        for(l=2, w=domainWidth-5; l<domainLength-7; l++)
                {
                switch(l)
                        {
                        case 9:
                        case 10:
                        case 11:
                                break;
                        default:
                                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                                SetPointType(timeSlice, w, l, OBJECT);
//                              SetAdjObjsInTimeSlice(timeSlice, w, l);
                                break;
                        }
                }
        for(w=domainWidth-5, l=domainLength-7; w<domainWidth; w++)
                {
                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                SetPointType(timeSlice, w, l, OBJECT);
//              SetAdjObjsInTimeSlice(timeSlice, w, l);
                }
        for(w=domainWidth-10, l=domainLength-2; w<domainWidth; w++)
                {
                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                SetPointType(timeSlice, w, l, OBJECT);
//              SetAdjObjsInTimeSlice(timeSlice, w, l);
                }
        }

void Domain::SetPointFrom(int timeSlice, int width, int length, int from)
        {
        POINT *tmp;

        if ((width < 0)
        || (length < 0)
        || (timeSlice < 0)
        || (width >= domainWidth)
        || (length >= domainLength)
        || (timeSlice >= domainTimeSlices))
                return;
        tmp = GetPoint(timeSlice, width, length);
        tmp->from = from;
        }

void Domain::SetPointType(int timeSlice, int width, int length, char type)
```

```
	{
	char oldType;
	POINT *tmp;

	if ((width < 0)
	|| (length < 0)
	|| (timeSlice < 0)
	|| (width >= domainWidth)
	|| (length >= domainLength)
	|| (timeSlice >= domainTimeSlices))
		return;
	tmp = GetPoint(timeSlice, width, length);
	oldType = tmp->type;
	tmp->type = type;
	if (oldType != OBJECT)
		{
//		SetAdjObjsInTimeSlice(timeSlice, width, length);
		}
	}


void Domain::SetStartFromFile(char*fileName)
	{
	int w, l;
	char recType;
	char tmp[256];
	fstream dataFile;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Start added.\n";

	for(;;)
		{
		dataFile. getline(tmp, sizeof(tmp));
		if (strlen(tmp) == 0)
			break;
		if (tmp[0] == START)
			{
//			cout << "The START line is: " << tmp << "\n";
			if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
				{
				SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
				SetPointType(domainTimeSlices-1, w, l, START);
				break;
				}
			else
				cout << "Improperly formatted line ignored\n";
			}
		}
	dataFile.close();
	}


void Domain::SetVerticesInTimeSlice(int t)
	{
	int w, l;
	int diff_counter, corner_counter;
	int side_bits, corner_bits;

	for(w=0; w<domainWidth; w++)
		{
		for(l=0; l<domainLength; l++)
			{
			if (!IsPointClear(t, w, l))
				continue;

			corner_bits = BITMASK_CLEAR;
			corner_counter = 0;
			if (IsPointNearObject(t, w-1, l-1))
				{
				corner_bits |= BITMASK_LEFT;
				corner_bits |= BITMASK_TOP;
				corner_counter++;
				}
			if (IsPointNearObject(t, w-1, l+1))
				{
				corner_bits |= BITMASK_LEFT;
				corner_bits |= BITMASK_BOTTOM;
				corner_counter++;
				}
			if (IsPointNearObject(t, w+1, l+1))
				{
				corner_bits |= BITMASK_RIGHT;
```

```
                                        corner_bits |= BITMASK_BOTTOM;
                                        corner_counter++;
                                        }
                                if (IsPointNearObject(t, w+1, l-1))
                                        {
                                        corner_bits |= BITMASK_RIGHT;
                                        corner_bits |= BITMASK_TOP;
                                        corner_counter++;
                                        }
                                if (corner_bits != BITMASK_CLEAR)
                                        {
                                        side_bits = BITMASK_CLEAR;
                                        if (IsPointNearObject(t, w-1, l))
                                                side_bits |= BITMASK_LEFT;
                                        if (IsPointNearObject(t, w, l-1))
                                                side_bits |= BITMASK_TOP;
                                        if (IsPointNearObject(t, w+1, l))
                                                side_bits |= BITMASK_RIGHT;
                                        if (IsPointNearObject(t, w, l+1))
                                                side_bits |= BITMASK_BOTTOM;

                                        for(diff_counter=0;
                                                (side_bits != BITMASK_CLEAR)
                                                || (corner_bits != BITMASK_CLEAR);
                                                side_bits >>= 1, corner_bits >>= 1)
                                                {
                                                if ((corner_bits & (int)0x01)
                                                && ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
                                                        {
                                                        diff_counter++;
                                                        continue;
                                                        }
                                                if ((side_bits & (int)0x01)
                                                && ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
                                                        {
                                                        diff_counter++;
                                                        continue;
                                                        }
                                                }
                                        if ((diff_counter > 2)
                                        || ((diff_counter == 2) && (corner_counter == 1)))
                                                {
                                                SetPointFrom(t, w, l, FROM_NOWHERE);
                                                SetPointType(t, w, l, VERTEX);
                                                }
                                        }
                                }
                        }
                }

// edge.hpp
typedef struct van
    {
        int t,w,l;
        float dist;
        struct van *prev, *next;
        } EDGE_NODE;

class EdgeList
        {
        public:
                EdgeList();
                ~EdgeList();
       EDGE_NODE* BuildNewEdge(int, int, int, float);
       void DelEdge(EDGE_NODE*);
                void DelEdgeToVertex(int, int, int);
       EDGE_NODE* GetFirstEdge(void);
       EDGE_NODE* GetNextEdge(EDGE_NODE*);
       void InsertNewEdge(EDGE_NODE*);
       void ListAllEdges(void);
   private:
        EDGE_NODE *edgeHead;
        };

// edge.cpp
#include <iostream.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include "edge.hpp"


EdgeList::EdgeList()
        {
```

```
	edgeHead = (EDGE_NODE*)NULL;
//	cout << "EdgeList Constructor\n";
	}


EdgeList::~EdgeList()
	{
	while (edgeHead != (EDGE_NODE*)NULL)
		DelEdge(edgeHead);
//	cout << "EdgeList Destructor\n";
	}


EDGE_NODE* EdgeList::BuildNewEdge(int t, int w, int l, float dist)
	{
	EDGE_NODE *a;

//	if (otherVertex != (void*)NULL)
//		{
		a = (EDGE_NODE*)malloc(sizeof(EDGE_NODE));
		if (a == (EDGE_NODE*)NULL)
			{
			cout << "Out of Memory in BuildNewEdge()\n";
			exit(0);
			}
		(*a).t = t;
		(*a).w = w;
		(*a).l = l;
		(*a).dist = dist;
		(*a).prev = (*a).next = (EDGE_NODE*)NULL;
		return(a);
//		}
//	return((EDGE_NODE*)NULL);
	}


void EdgeList::DelEdge(EDGE_NODE*a)
  {
  EDGE_NODE*tmp;

//	cout << "edge 0\n";
	if (a != (EDGE_NODE*)NULL)
		{
		if ((a->prev == (EDGE_NODE*)NULL)  // del last remaining edge
		&& (a->next == (EDGE_NODE*)NULL))
			{
//			cout << "edge 1\n";
			free(a);
			edgeHead = (EDGE_NODE*)NULL;
			return;
			}
		if ((a->prev != (EDGE_NODE*)NULL)  // del edge in middle
		&& (a->next != (EDGE_NODE*)NULL))
			{
//			cout << "edge 2\n";
			tmp = a->prev;
			tmp->next = a->next;
			tmp = a->next;
			tmp->prev = a->prev;
			free(a);
			return;
			}
		if ((a->prev == (EDGE_NODE*)NULL)  // del edge at sol
		&& (a->next != (EDGE_NODE*)NULL))
			{
//			cout << "edge 3\n";
			tmp = a->next;
			tmp->prev = (EDGE_NODE*)NULL;
			edgeHead = tmp;
			free(a);
			return;
			}
		if ((a->prev != (EDGE_NODE*)NULL)  // del edge at eol
		&& (a->next == (EDGE_NODE*)NULL))
			{
//			cout << "edge 4\n";
			tmp = a->prev;
			tmp->next = (EDGE_NODE*)NULL;
			free(a);
			return;
			}
		}
	}
```

```
void EdgeList::DelEdgeToVertex(int t, int w, int l)
	{
	EDGE_NODE*tmp;

	for(tmp = GetFirstEdge(); tmp != (EDGE_NODE*)NULL; tmp = GetNextEdge(tmp))
		{
		if ((tmp->t == t)
	&& (tmp->w == w)
	&& (tmp->l == l))
			{
			DelEdge(tmp);
			break;
			}
		}
	}


EDGE_NODE* EdgeList::GetFirstEdge()
	{
   return(edgeHead);
   }


EDGE_NODE* EdgeList::GetNextEdge(EDGE_NODE*cur)
   {
   return(cur->next);
   }


void EdgeList::InsertNewEdge(EDGE_NODE *edge)
	{
	EDGE_NODE*cur;

	if (edge == (EDGE_NODE*)NULL)
		return;

	if (edgeHead == (EDGE_NODE*)NULL)
		{
		edgeHead = edge;
		edge->prev = edge->next = (EDGE_NODE*)NULL;
//		cout << "Inserted edge into empty list ";
//		cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//		cout << "Dist = " << edge->dist << "\n";
		return;
		}

	for(cur=edgeHead; cur != (EDGE_NODE*)NULL; cur=(*cur).next)
		{
		if ( (edge->t > cur->t)
		|| ((edge->t == cur->t)
			&& (edge->w > cur->w))
		|| ((edge->t == cur->t)
			&& (edge->w == cur->w)
			&& (edge->l > cur->l)) )
			{					// insert after cur
			if (cur->next == (EDGE_NODE*)NULL)
				{
				edge->next = (EDGE_NODE*)NULL;// no more so append to eol
				edge->prev = cur;
				cur->next = edge;
				break;
				}
			else
				continue;			// try next one
			}
		if ( (edge->t < cur->t)
		|| ((edge->t == cur->t)
			&& (edge->w < cur->w))
		|| ((edge->t < cur->t)
			&& (edge->w == cur->w)
			&& (edge->l < cur->l)) )
			{					// insert before cur
			if (cur->prev == (EDGE_NODE*)NULL)
				{
				edge->prev = (EDGE_NODE*)NULL;	// at start of list
				edge->next = cur;
				cur->prev = edge;
				edgeHead = edge;
				break;
				}
			else
				{
				edge->prev = cur->prev;		// in middle/end of list
```

```
                        edge->next = cur;
                        cur->prev = edge;
                        cur = edge->prev;
                        cur->next = edge;
                        break;
                        }
                }
        if ( (edge->t == cur->t)
        && (edge->w == cur->w)
        && (edge->l == cur->l) )
                {
                /* already here - replace it ! */
                cout << "Already here - replacing values and disposing of new edge ";
    cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
    cout << "Dist = " << edge->dist << "\n";
                cur->w = edge->w;
                cur->l = edge->l;
                cur->dist = edge->dist;
                free(edge);
                break;
                }
        }
//      cout << "Inserted edge into list ";
//      cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//      cout << "Dist = " << edge->dist << "\n";
        }


void EdgeList::ListAllEdges()
        {
        EDGE_NODE *edge;

        for(edge = edgeHead; edge != (EDGE_NODE*)NULL; edge = (*edge).next)
                {
                cout << "Edge to (";
                cout << (*edge).t << ",";
                cout << (*edge).w << ",";
                cout << (*edge).l << "). Dist is " << (*edge).dist << "\n";
                }
        cout << "-----------------\n";
        }

// object.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define NUM_DIRS 7 /* this is the 8 horizontal directions; (0->7) */
enum directions { front, back, left, right, frontright, frontleft, backright, backleft, up, down, clear};


typedef struct o
        {
        int direction, velocity;
        int w, l;
        struct o *prev, *next;
        } OBJECT_NODE;


class ObjectList
        {
        public:
                ObjectList();
                ~ObjectList();
                OBJECT_NODE* BuildNewObject(int, int);
        void DelAllObjects(void);
        void DelObject(OBJECT_NODE*);
        OBJECT_NODE* GetNextObject(void);
                void InsertNewObject(OBJECT_NODE*);
        void ListAllObjects(void);
        private:
                OBJECT_NODE *objectHead;
        };

//object.cpp
#include <iostream.h>
```

```
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
// #include "edge.hpp"
#include "object.hpp"

ObjectList::ObjectList()
        {
        objectHead = (OBJECT_NODE*)NULL;
        cout << "ObjectList Constructor\n";
        }



ObjectList::~ObjectList()
        {
        cout << "ObjectList Destructor\n";
        DelAllObjects();
        }



OBJECT_NODE* ObjectList::BuildNewObject(int w, int l)
        {
        OBJECT_NODE *newPtr;

        newPtr = (OBJECT_NODE*)malloc(sizeof(OBJECT_NODE));
        if (newPtr == (OBJECT_NODE*)NULL)
                {
                cout << "Out of memory in BuildNewObject()\n";
                return(NULL);
                }
        (*newPtr).direction = rand() % NUM_DIRS;
        (*newPtr).velocity = 1;
        (*newPtr).prev = (*newPtr).next = (OBJECT_NODE*)NULL;
        (*newPtr).w = w;
        (*newPtr).l = l;
        return(newPtr);
        }



void ObjectList::DelAllObjects()
        {
        OBJECT_NODE *tmp;

        while (objectHead != (OBJECT_NODE*)NULL)
                {
                tmp = objectHead;
                objectHead = (*objectHead).next;
                DelObject(tmp);
                }
        }



void ObjectList::DelObject(OBJECT_NODE *todie)
   {
        OBJECT_NODE *cur;

        if (((*todie).prev != (OBJECT_NODE*)NULL)
        && ((*todie).next != (OBJECT_NODE*)NULL))
                {
//              cout << "Deleted Object (" << todie->w << "," << todie->l << ")\n";
                cur=(*todie).prev;
                if (cur == todie)
                        {
                        objectHead = (OBJECT_NODE*)NULL;
//                      cout << "Object List Empty\n";
                        free(todie);
                        return;
                        }
                else
                        {
                        (*cur).next=(*todie).next;
                        cur = (*todie).next;
                        (*cur).prev = (*todie).prev;
                        if (objectHead == todie)
                                objectHead = (*todie).next;
        free(todie);
          return;
                        }
                }
        cout << "***Did NOT delete rotton Object (" << todie->w << "," << todie->l << ")\n";
```

```
        }

OBJECT_NODE* ObjectList::GetNextObject()
    {
    OBJECT_NODE*tmp;

        if (objectHead == (OBJECT_NODE*)NULL)
               return((OBJECT_NODE*)NULL);

        tmp = objectHead;
    objectHead = objectHead->next;
    return(tmp);
    }

void ObjectList::InsertNewObject(OBJECT_NODE*newPtr)
        {
        OBJECT_NODE *cur;

        if (newPtr == (OBJECT_NODE*)NULL)
               return;

        if (objectHead == (OBJECT_NODE*)NULL)
               {
               objectHead = newPtr;
               (*newPtr).prev = (*newPtr).next = newPtr;
               }
        else
               {
               /* insert before first node */
               cur = objectHead;
               (*newPtr).next = cur;
               (*newPtr).prev = (*cur).prev;
               (*cur).prev = newPtr;

               cur = (*newPtr).prev;
               (*cur).next = newPtr;
               objectHead = newPtr;
        }
//      cout << "Inserted object (" << newPtr->w << "," << newPtr->l << ")\n";
        }


void ObjectList::ListAllObjects()
        {
    OBJECT_NODE*tmp;

    if (objectHead == (OBJECT_NODE*)NULL)
        {
        cout << "ObjectList is empty\n";
        return;
        }
    for(tmp=objectHead;;tmp = tmp->next)
        {
        cout << "Object:(" << tmp->w << "," << tmp->l << ")\n";
        tmp = tmp->next;
        if (tmp == objectHead)
           {
           cout << "ObjectList ended\n";
           break;
           }
        }
        }

// vertex.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00
```

```
typedef struct vn
        {
        int t,w,l;
        char nodeType;
        struct vn *searchPrev, *searchNext;
        float searchDist;
        int searchMarker;
        EdgeList *edgeList;
        struct vn *pathFrom, *pathTo;
        struct vn *prev, *next;
        } VERTEX_NODE;


class VertexList
        {
        public:
                VertexList();
                ~VertexList();
                void AddToSearchList(VERTEX_NODE*);
                VERTEX_NODE* BuildNewVertex(int,int,int,char);
                int CalcRobotDir(VERTEX_NODE*);
                void DelAllVertices(void);
        void DelVertex(VERTEX_NODE*);
        int FindPath(void);
        VERTEX_NODE* FindVertex(int, int, int);
                VERTEX_NODE* GetFirstVertex(void);
        VERTEX_NODE* GetGoalVertex(void);
                VERTEX_NODE* GetNextVertex(VERTEX_NODE*);
        VERTEX_NODE* GetStartVertex(void);
        void InsertAllVertices(void);
                void InsertNewVertex(VERTEX_NODE*);
        void ListAllVertices(void);
        void ListSearchList(void);
//              void MoveRobot(int, int, int, int);
                void MarkPath(VERTEX_NODE*);
                void RemoveFromSearchList(VERTEX_NODE*);
                void TrimSearchList(void);
        private:
                VERTEX_NODE* vertexHead;
        VERTEX_NODE* searchHead;
        int searchMarker;
        float searchTrimDist;
        };

// vertex.cpp
#include <iostream.h>
#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include "edge.hpp"
#include "vertex.hpp"

VertexList::VertexList()
        {
        vertexHead = (VERTEX_NODE*)NULL;
   searchHead = (VERTEX_NODE*)NULL;
   searchMarker = 0;
   searchTrimDist = -1.0;
   cout << "Initialised Vertex class\n";
        }



VertexList::~VertexList()
        {
        cout << "VertexList destructor started\n";
        DelAllVertices();
        cout << "VertexList destructor ended!\n";
        }


void VertexList::AddToSearchList(VERTEX_NODE*a)
        {
        VERTEX_NODE *tmp;

        if (a == (VERTEX_NODE*)NULL)
                return;

        if (((*a).searchPrev != (VERTEX_NODE*)NULL) // if already in fringe
        || ((*a).searchNext != (VERTEX_NODE*)NULL)) // list, dont add again
                return;
```

```
        if (searchHead == (VERTEX_NODE*)NULL)
                {
                searchHead = a;
                a->pathFrom = a->pathTo = (VERTEX_NODE*)NULL;
                a->searchPrev = a->searchNext = a;
                return;
                }

    for (tmp = searchHead;tmp != (VERTEX_NODE*)NULL;)
        {
                if (a->searchDist <= tmp->searchDist)
                        {
                        a->searchPrev = tmp->searchPrev;
                        a->searchNext = tmp;
                        tmp->searchPrev = a;
                        tmp = a->searchPrev;
                        tmp->searchNext = a;
                        if (searchHead == a->searchNext)
                                searchHead = a;
                        return;
                        }
                if (tmp->searchNext == searchHead)
                        {
                        a->searchPrev = tmp;
                        a->searchNext = searchHead;
                        tmp->searchNext = searchHead->searchPrev = a;
                        return;
                        }
                else
                        tmp = tmp->searchNext;
                }
        }


VERTEX_NODE* VertexList::BuildNewVertex(int t, int w, int l, char nodeType)
        {
        VERTEX_NODE *newPtr;

        newPtr = (VERTEX_NODE*)malloc(sizeof(VERTEX_NODE));
        if (newPtr == (VERTEX_NODE*)NULL)
                {
                cout << "Malloc() failed in BuildNewVertex!\n";
                exit(0);
                }
        (*newPtr).t = t;
        (*newPtr).w = w;
        (*newPtr).l = l;
        (*newPtr).nodeType = nodeType;
        (*newPtr).edgeList = new EdgeList();
        (*newPtr).pathTo = (*newPtr).pathFrom = (VERTEX_NODE*)NULL;
        (*newPtr).searchDist = 0.0;
        (*newPtr).searchPrev = (*newPtr).searchNext = (VERTEX_NODE*)NULL;
        (*newPtr).prev = (*newPtr).next = (VERTEX_NODE*)NULL;
        if (nodeType == START)
                (*newPtr).searchMarker = searchMarker+1;
        else
                (*newPtr).searchMarker = searchMarker;
        return(newPtr);
        }


void VertexList::DelAllVertices()
   {
        while (vertexHead != (VERTEX_NODE*)NULL)
                {
//              cout << "About to delete (" << vertexHead->t << "," << vertexHead->w ;
//              cout << "," << vertexHead->l << ") at addr: " << vertexHead ;
//              cout << ": Prev:" << vertexHead->prev << ": Next:" << vertexHead->next <<"\n";
                DelVertex(vertexHead);
                }
        cout << "----------\n";
//      getch();
        }


void VertexList::DelVertex(VERTEX_NODE *todie)
        {
        VERTEX_NODE *tmp;
        EDGE_NODE*a;

        if (todie == (VERTEX_NODE*)NULL)
                return;
```

```
         for(a = todie->edgeList->GetFirstEdge();
                 a != (EDGE_NODE*)NULL;
                 a = todie->edgeList->GetFirstEdge())
                 {
        todie->edgeList->DelEdgeToVertex(a->t, a->w, a->l);
        tmp = FindVertex(a->t, a->w, a->l);
        if (tmp != (VERTEX_NODE*)NULL)
           {
           tmp->edgeList->DelEdgeToVertex(todie->t, todie->w, todie->l);
           }
//               tmp = (VERTEX_NODE*)(a->otherVertex);
//               cout << "On vertex (" << todie->t << "," << todie->w << ",";
//               cout << todie->l << "):Del edge to (" << tmp->t << "," << tmp->w << ",";
//               cout << tmp->l << ")\n";
//               cout << "todie at:" << todie << ". a at:" << a << "\n";
//               cout << "  tmp at:" << tmp << "\n";
//               todie->edgeList->DelEdgeToVertex((void*)tmp);
//               cout << "On vertex (" << tmp->t << "," << tmp->w << ",";
//               cout << tmp->l << "):Del edge to (" << todie->t << "," << todie->w << ",";
//               cout << todie->l << ")\n";
//               tmp->edgeList->DelEdgeToVertex((void*)todie);
                 }
        delete todie->edgeList;
        RemoveFromSearchList(todie);

        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
                {
                vertexHead = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
                tmp = (*todie).prev;
                (*tmp).next = (*todie).next;
                tmp = (*todie).next;
                (*tmp).prev = (*todie).prev;
                free(todie);
                return;
                }
        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
                vertexHead = tmp = (*todie).next;
                (*tmp).prev = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
                {
                tmp = (*todie).prev;
                (*tmp).next = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        }


int VertexList::FindPath()
   {
        EDGE_NODE *e;
        int goalFound=FALSE;
        float dist;
        VERTEX_NODE *cur, *adj, *dest;

        if (searchHead->nodeType == START)
               searchMarker = searchHead->searchMarker;
        else
               return(FALSE);

   dest = GetGoalVertex();
   if (dest == (VERTEX_NODE*)NULL)
      return(FALSE);

        for (cur = searchHead; cur != (VERTEX_NODE*)NULL; cur = searchHead)
               {
               if ((cur->searchDist >= searchTrimDist)
               && (searchTrimDist > 0.0))
                      {
                      RemoveFromSearchList(cur);
```

```
                    continue;
                    }
               for(e = cur->edgeList->GetFirstEdge();
                    e != (EDGE_NODE*)NULL;
                    e = cur->edgeList->GetNextEdge(e))
                    {
          adj = FindVertex(e->t, e->w, e->l);
          if (adj == (VERTEX_NODE*)NULL)
             continue;

//          adj = (VERTEX_NODE*) e->otherVertex;
                    dist = cur->searchDist + e->dist;

          if ( (adj->searchMarker != searchMarker)
                    || ((adj->searchMarker == searchMarker)
                         && (adj->searchDist > dist)) )
                         {
                         if ((dist < searchTrimDist)
                         || (searchTrimDist <= 0.0))
                              {
                              adj->pathFrom = cur;
                              adj->searchMarker = searchMarker;
                              adj->searchDist = dist;
                              AddToSearchList(adj);
                              if (adj->nodeType == GOAL)
                                   {
                                   goalFound = TRUE;
                                   searchTrimDist = dist;
                                   MarkPath(adj);
                                   TrimSearchList();
                                   }
                              }
                         }
                    }
               RemoveFromSearchList(cur);
               }
        return(goalFound);
        }


VERTEX_NODE* VertexList::FindVertex(int t, int w, int l)
   {
   VERTEX_NODE* cur;

   for(cur = GetFirstVertex();
      cur != (VERTEX_NODE*)NULL;
      cur = GetNextVertex(cur))
      {
      if ((cur->t == t)
      && (cur->w == w)
      && (cur->l == l))
         return(cur);          // found it

      if ((cur->t >= t)
      && (cur->w >= w)
      && (cur->l >= l))
         break;                // passed it - it's not in the list
      }
   return((VERTEX_NODE*)NULL);
   }


VERTEX_NODE* VertexList::GetFirstVertex()
   {
   return(vertexHead);
   }


VERTEX_NODE* VertexList::GetGoalVertex()
   {
   VERTEX_NODE*tmp;

   for(tmp = GetFirstVertex();
      tmp != (VERTEX_NODE*)NULL;
      tmp = GetNextVertex(tmp))
      {
            if (tmp->nodeType == GOAL)
         break;
      }
   return(tmp);
   }
```

```
VERTEX_NODE* VertexList::GetNextVertex(VERTEX_NODE*cur)
   {
   return(cur->next);
   }


VERTEX_NODE* VertexList::GetStartVertex()
   {
   VERTEX_NODE*tmp;

   for(tmp = GetFirstVertex();
      tmp != (VERTEX_NODE*)NULL;
      tmp = GetNextVertex(tmp))
      {
            if (tmp->nodeType == START)
         break;
      }
   return(tmp);
   }


void VertexList::InsertNewVertex(VERTEX_NODE*a)
       {
       VERTEX_NODE *cur;

       if (a == (VERTEX_NODE*)NULL)
              return;

       if (vertexHead == (VERTEX_NODE*)NULL)
              {
              vertexHead = a;
              (*a).prev = (*a).next = (VERTEX_NODE*)NULL;
              return;
              }

       for(cur=vertexHead; cur != (VERTEX_NODE*)NULL;   cur=(*cur).next)
              {
              if ( (a->t > cur->t)
      || ((a->t == cur->t)
         && (a->w > cur->w))
       || ((a->t == cur->t)
         && (a->w == cur->w)
              && (a->l > cur->l)) )
                     {                                // insert after cur
                     if ((*cur).next == (VERTEX_NODE*)NULL)
                            {
                            (*a).next = (*cur).next;     // eol - append new node
                            (*a).prev = cur;
                            (*cur).next = a;
                            break;
                            }
                     else
                            continue;                    // get next node
                     }
              if ( (a->t < cur->t)
      || ((a->t == cur->t)
         && (a->w < cur->w))
      || ((a->t == cur->t)
                 && (a->w == cur->w)
                     && (a->l < cur->l)) )
                     {                                // insert before cur
                     if (cur->prev == (VERTEX_NODE*)NULL)
                            {
                            a->prev = (VERTEX_NODE*)NULL;// at start of list
                            a->next = cur;
                            cur->prev = a;
                            vertexHead = a;
                            break;
                            }
                     else
                            {
                            a->prev = cur->prev;          // in middle/end of list
                            a->next = cur;
                            cur->prev = a;
                            cur = a->prev;
                            cur->next = a;
                            break;
                            }
                     }
              if ((a->t == cur->t)
      && (a->w == cur->w)
              && (a->l == cur->l))
                     {                                // insert after cur at eol
                     if ((*cur).next == (VERTEX_NODE*)NULL)
```

```
                                {
                                (*a).next = (VERTEX_NODE*)NULL;
                                (*a).prev = cur;
                                (*cur).next = a;
                                break;
                                }
                        else
                                continue;
                        }
                }
//      cout << "Inserted vertex (";
// cout << (*a).t << "," << (*a).w << "," << (*a).l << ")\n";
        }


void VertexList::ListAllVertices()
        {
        VERTEX_NODE *cur=vertexHead;

    while (cur != (VERTEX_NODE*)NULL)
        {
                cout << "Vertex:" <<  (*cur).nodeType << ": at (";
                cout << (*cur).t << "," << (*cur).w << "," << (*cur).l << ")\n";
                cur->edgeList->ListAllEdges();
        cur = (*cur).next;
        }
    }


void VertexList::ListSearchList()
        {
        VERTEX_NODE*cur;

        if (searchHead == (VERTEX_NODE*)NULL)
                {
                cout << "Empty SearchList! (marker=" << searchMarker << ")\n";
                return;
                }

        cout << "List of nodes in Search list (marker=" << searchMarker << ")\n";
        cur = searchHead;
        do
                {
                cout << "SearchList node (" << cur->t << "," << cur->w << "," << cur->l << ") ";
        cout << "Dist = " << cur->searchDist << "\n";
                cur=cur->searchNext;
                }
        while (cur != searchHead);
        cout << "End of SearchList\n";
        }


void VertexList::MarkPath(VERTEX_NODE*v)
        {
        VERTEX_NODE*tmp;

        if (v == (VERTEX_NODE*)NULL)
                return;

//      cout << "Path from Goal to Start\n";
        v->pathTo = (VERTEX_NODE*)NULL;
        do
                {
                tmp = v->pathFrom;
//              cout << "(" << v->t << "," << v->w << "," << v->l << ")\n";
                if (tmp != (VERTEX_NODE*)NULL)
                        {
                        tmp->pathTo = v;
                        v = v->pathFrom;
                        }
                }
        while (tmp != (VERTEX_NODE*)NULL);
        }


void VertexList::RemoveFromSearchList(VERTEX_NODE*a)
        {
        VERTEX_NODE *cur;

        if ((a->searchPrev == (VERTEX_NODE*)NULL)
        || (a->searchNext == (VERTEX_NODE*)NULL))
       {
       if (searchHead == a)
          searchHead = (VERTEX_NODE*)NULL;
```

```
        return;
        }

         cur=a->searchPrev;
         if (cur == a)
                {
        searchHead = (VERTEX_NODE*)NULL;
//              cout << "Removed Search Node (";
//      cout << a->t << "," << a->w << "," << a->l << "). SearchList empty\n";
                }
         else
                {
                cur->searchNext = a->searchNext;
                cur = a->searchNext;
                cur->searchPrev = a->searchPrev;
                if (searchHead == a)
                        searchHead = a->searchNext;
//              cout << "Removed Search Node (";
//      cout << a->t << "," << a->w << "," << a->l << ").\n";
                }
        a->searchPrev = a->searchNext = (VERTEX_NODE*)NULL;
        }

void VertexList::TrimSearchList()
   {
        VERTEX_NODE *cur, *tmpPtr=(VERTEX_NODE*)0;
        int trimmed=TRUE;

//      cout << "Trimming search list to " << searchTrimDist << " or less\n";
        if ((searchHead == (VERTEX_NODE*)NULL)
   || (searchTrimDist < 0.0))
                return;

   cur = searchHead;
        while ((tmpPtr != cur) || (trimmed == TRUE))
                {
                if (trimmed==TRUE)
                        {
                        tmpPtr=cur;
                        trimmed=FALSE;
                        }
                if ((*cur).searchDist >= searchTrimDist)
                        {
//                      cout << "Trimmed out (";
//              cout << cur->t << "," << cur->w << "," << cur->l << ") ";
//              cout << "Dist was " << cur->searchDist << "\n";
                        if (cur == cur->searchNext)
                                {
          RemoveFromSearchList(cur);
                                break;
                                }
                        else
                                {
                                tmpPtr = cur->searchNext;
                                RemoveFromSearchList(cur);
                                cur = tmpPtr;
                                trimmed = TRUE;
                                }
                        }
                else
                        cur = cur->searchNext;
                }
//   cout << "-------\n";
   }
```

B.6. Standard A* Graph Theory

The Standard A* Search Algorithm for Graph Theory which was implemented as part of this project was coded in ANSI C++. The filenames for each of the separate source code files were supplied inside C++ format comments (i.e. //) at the beginning of each file listing. A detailed explanation of the design behind this program was presented in Chapter 4.

```
// main.cpp
#include <iostream.h>
```

```
#include <fstream.h>
#include <stdio.h>
#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <sys\timeb.h>
#include <dir.h>
#include <ctype.h>
#include "edge.hpp"
#include "vertex.hpp"
#include "object.hpp"
#include "domain.hpp"
#include "bench.hpp"

#define MAPFILE_MASK "MAP*.DAT"

int main(int, char**);
void BuildVertexListFromDomain(VertexList&, Domain&);
int CalcRobotDir(VERTEX_NODE*);
void FindAllEdges(VertexList&, Domain&);
void MemStatus(char*);
int SetWorkingDir(void);
void RenameMapFile(char*);


int main(int argc, char** argv)
	{
	char* shortName;
	struct ffblk mapfile;

	for(;;)
		{
		shortName = strstr(argv[0], "\\");
		if (shortName == (char*)NULL)
			{
			shortName = argv[0];
			cout << "This is " << shortName << "\n";
			break;
			}
		else
			argv[0] = shortName+1;
		}

	MemStatus("Free memory before mainloop in main: ");

	if (!SetWorkingDir())
		{
		cout << "Program exiting gracefully\n";
		return(1);
		}
	if (findfirst(MAPFILE_MASK, &mapfile, 0))
		{
		cout << "Program exiting gracefully - no map files found\n";
		return(1);
		}
	do
		{
		char tmp[256];
		int i;
		Benchmark stopWatch(shortName);

		sprintf(tmp, "Starting on %s: ", mapfile.ff_name);
		MemStatus(tmp);
		for (i=0; i<NUMTIMES; i++)
			{
			int flag, timeTaken;
			float distTravelled;
			Domain theWorld(10, 20, 20, mapfile.ff_name);

			timeTaken = 0;
			distTravelled = 0.0;
			stopWatch.IterStart(i, mapfile.ff_name);
			stopWatch.Click();
			for(;;)
				{
				VertexList vertList;

				BuildVertexListFromDomain(vertList, theWorld);
				FindAllEdges(vertList, theWorld);
				stopWatch.Click();
				theWorld.DrawDomain();
				stopWatch.Click();
				flag = vertList.FindPath();
```

```
                                        stopWatch.Click();
                                        if (flag == TRUE)
                                                cout << "Path found the GOAL!\n";
                                        else
                                                cout << "No path found to GOAL!\n";

                                        if (!flag)
                                                {
                                                VERTEX_NODE*t1, *t2;

                                                t1 = vertList.GetStartVertex();
                                                if (t1 != (VERTEX_NODE*)NULL)
                                                        {
                                                        t2 = t1->pathTo;
                                                        if (t2 != (VERTEX_NODE*)NULL)
                                                                {
                                                                if ((t1->w == t2->w)
                                                                && (t1->l == t2->l)
                                                                && (t2->nodeType == GOAL))
                                                                        {
                                                                        cout << "MADE IT TO THE GOAL!\n";
                                                                        break;
                                                                        }
                                                                distTravelled += theWorld.MoveRobot(t1->t, t1->w, t1->l,

                                t2->t, t2->w, t2->l);
                                                                }
                                                        }
                                                }
                                        theWorld.AdvanceTime();
                                        timeTaken++;
                                        }
                                stopWatch.IterStop(timeTaken, distTravelled);
                                }
                        stopWatch.LogCalcs();
                        RenameMapFile(mapfile.ff_name);
//                      break;
                        }
        while (!findnext(&mapfile));

        MemStatus("Free memory after mainloop in main: ");
        return(0);
        }


void BuildVertexListFromDomain(VertexList &vList, Domain &domain)
        {
        char type;
        int t=domain.GetDomainTimeSlices(), a;
        int w=domain.GetDomainWidth(), b;
        int l=domain.GetDomainLength(), c;
        VERTEX_NODE*newPtr, *gPtr, *sPtr;

        for(gPtr = sPtr = (VERTEX_NODE*)NULL, a=0; a<t; a++)
          {
                for(b=0; b<w; b++)
                        {
                        for(c=0; c<l; c++)
                                {
                                type = domain.GetPointType(a, b, c);
                                if ((type == VERTEX)
                                || (type == START)
                                || (type == GOAL))
                                        {
                                        newPtr = vList.BuildNewVertex(a, b, c, type);
                                        vList.InsertNewVertex(newPtr);
                                        }
             if (type == GOAL)
                gPtr = newPtr;
             if (type == START)
                sPtr = newPtr;
                                }
                        }
                }
   if ((gPtr != (VERTEX_NODE*)NULL)
   && (sPtr != (VERTEX_NODE*)NULL))
      {
      sPtr->searchDist = 0.0;
             vList.AddToSearchList(sPtr);
      }
        }


void FindAllEdges(VertexList &vList, Domain&domain)
```

```
	{
	EDGE_NODE*e;
	VERTEX_NODE*a,*b;
	float dist;

	for(a = vList.GetFirstVertex();
		a != (VERTEX_NODE*)NULL;
		a = vList.GetNextVertex(a))
		{
		for(b = vList.GetNextVertex(a);
			b != (VERTEX_NODE*)NULL;
			b = vList.GetNextVertex(b))
	  {
	  if (a == b)
				continue;

			if ((a->t == b->t)
			&& (a->w == b->w)
			&& (a->l == b->l))
				continue;

	  dist = domain.CheckLine(a->t, a->w, a->l, b->t, b->w, b->l);
	  if (dist > 0.0)
	     {
				e = a->edgeList->BuildNewEdge(b->t, b->w, b->l, dist);
				a->edgeList->InsertNewEdge(e);
				}
			dist = domain.CheckLine(b->t, b->w, b->l, a->t, a->w, a->l);
			if (dist > 0.0)
				{
				e = b->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
				b->edgeList->InsertNewEdge(e);
	     }
	  }
	     }
	}


void MemStatus(char *StatusMessage)
	{
	char tmp[256];
	fstream debugFile;
	long MemLeft;

	debugFile.open("DEBUG.LOG", ios::app);
	if (!debugFile)
		cout << "Unable to open DEBUG.LOG\n";

	MemLeft = (long) coreleft();
	sprintf(tmp, "%s%ld\n", StatusMessage, MemLeft);
	debugFile.write(tmp, strlen(tmp));
	debugFile.close();
//	cout << StatusMessage << MemLeft << "\n";
	}


int SetWorkingDir()
	{
	char mapdir[256];

	cout <<"Enter the directory containing map files, or \"q\" for quit:";
	cin >> mapdir;
	if (toupper(mapdir[0]) == 'Q')
		{
		cout << "Quitting...\n";
		return(FALSE);
		}
	if (chdir(mapdir))
		{
		cout << "The directory " << mapdir << " could not be found.\n";
		return(FALSE);
		}
	cout << "Made " << mapdir << " the current directory.\n";
	return(TRUE);
	}


void RenameMapFile(char*filename)
	{
	char newfilename[128];
	char* ch;
	int i=(int)'.';

	strcpy(newfilename, filename);
```

```
	ch = strrchr(newfilename,i);
	if (ch != (char*)NULL)
		{
		strcpy(ch, ".bak");
		rename(filename, newfilename);
		}
	else
		exit(1);
	}

// bench.hpp
#define FALSE 0
#define TRUE !FALSE
#define NUMTIMES 10
#define MAXFILENAME 13

class Benchmark
	{
	public:
		Benchmark(char*);
		~Benchmark();
		void Click(void);
		void IterStart(int, char*);
		void IterStop(int, float);
		void LogCalcs(void);
	private:
		void Diff(struct timeb*, struct timeb*, struct timeb*);

		fstream logFile;
		fstream avgFile;
		char mapFileName[MAXFILENAME];
		struct timeb benchmarks[NUMTIMES][2];
		float distRobotTravelled[NUMTIMES];
		int timeTaken[NUMTIMES];
		int currentIter;
		int clickToggleFlag;
		struct timeb elapsedTime, computeTime;
		struct timeb startTime, clickOnTime, clickOffTime;
	};

//bench.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <alloc.h>		// for coreleft()
#include <stdlib.h>		// for itoa()
#include <string.h>		// for strcpy()
#include <sys\timeb.h>
#include "bench.hpp"


Benchmark::Benchmark(char *fileName)
	{
	char *tmp;
	char logname[MAXFILENAME];

	while((tmp = strchr(fileName, '\\')) != (char*)NULL)
		fileName = tmp+1;
	strcpy(logname, fileName);
	if ((tmp = strchr(logname, '.')) != (char*)NULL)
		strcpy(tmp, ".LOG");
	else
		strcat(logname, ".LOG");
	logFile.open(logname, ios::app);
	if (!logFile)
		cout << "Unable to open " << logname << "\n";

	if ((tmp = strchr(logname, '.')) != (char*)NULL)
		strcpy(tmp, ".AVG");
	else
		strcat(logname, ".AVG");
	avgFile.open(logname, ios::app);
	if (!avgFile)
		cout << "Unable to open " << logname << "\n";

	cout << "Initialised Benchmark class\n";
	}


Benchmark::~Benchmark()
	{
	if (logFile)
		{
		logFile.flush();
```

```
                logFile.close();
                }
        if (avgFile)
                {
                avgFile.flush();
                avgFile.close();
                }
        cout << "Closed log files and Destroying Benchmark class\n";
        }


void Benchmark::Click()
        {
        switch(clickToggleFlag)
                {
                case TRUE:
                        ftime(&clickOnTime);
                        clickToggleFlag = FALSE;
                        break;
                case FALSE:
                default:
                        ftime(&clickOffTime);
                        Diff(&clickOnTime, &clickOffTime, &computeTime);
                        clickToggleFlag = TRUE;
                }
        }


void Benchmark::Diff(struct timeb*start, struct timeb*stop, struct timeb*diff)
        {
        if ((*stop).millitm < (*start).millitm)
                {
                (*stop).millitm += (short)1000; /* carry when subtracting, stops*/
                (*start).time += 1L;            /* negative wraparound problems!*/
                }
        (*diff).millitm += (*stop).millitm - (*start).millitm;
        (*diff).time += (long)((*diff).millitm / (short)1000);
        (*diff).millitm %= (short)1000;
        (*diff).time += ((*stop).time-(*start).time);
        }


void Benchmark::IterStart(int i, char*s)
        {
        currentIter = i;
        clickToggleFlag = TRUE;
        computeTime.time = elapsedTime.time = 0L;
        computeTime.millitm = elapsedTime.millitm = 0;

        strcpy(mapFileName, s);

        ftime(&startTime);
        }


void Benchmark::IterStop(int t, float distTravelled)
        {
        struct timeb stopTime;

        ftime(&stopTime);
        Diff(&startTime, &stopTime, &elapsedTime);

        benchmarks[currentIter][0].time = elapsedTime.time;
        benchmarks[currentIter][0].millitm = elapsedTime.millitm;
        benchmarks[currentIter][1].time = computeTime.time;
        benchmarks[currentIter][1].millitm = computeTime.millitm;
        timeTaken[currentIter] = t;
        distRobotTravelled[currentIter] = distTravelled;
        }



void Benchmark::LogCalcs()
        {
        char tmp[256];
        float avgDist;
        struct timeb avg;
        int i, avgTimeTaken;

        for(i=0, avg.time=0L, avg.millitm=0, avgDist=0.0, avgTimeTaken=0;
                i<NUMTIMES;
                i++)
                {
```

```
            sprintf(tmp, "%s (%02d) Elapsed time:%05ld.%03d\n", mapFileName, i,
                                benchmarks[i][0].time, benchmarks[i][0].millitm);
            logFile.write(tmp, strlen(tmp));
            sprintf(tmp, "%s (%02d) Compute time:%05ld.%03d\n", mapFileName, i,
                                benchmarks[i][1].time, benchmarks[i][1].millitm);
            logFile.write(tmp, strlen(tmp));
            sprintf(tmp, "%s (%02d) Dist travelled:%f\n", mapFileName, i,
                                distRobotTravelled[i]);
            logFile.write(tmp, strlen(tmp));
            sprintf(tmp, "%s (%02d) Time Slices Taken:%d\n", mapFileName, i,
                                timeTaken[i]);
            logFile.write(tmp, strlen(tmp));
            avg.time+=benchmarks[i][0].time;
            avg.millitm+=benchmarks[i][0].millitm;
            if (avg.millitm % 1000 != avg.millitm)
                  {
                  avg.time += (long)(avg.millitm / (short)1000);
                  avg.millitm %= (short)1000;
                  }
            }

      sprintf(tmp, "%s Tot. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
      logFile.write(tmp, strlen(tmp));
      cout << avg.time << "." << avg.millitm << "\n";
      cout << tmp << "\n";

      i = (int) (avg.time % (long)NUMTIMES);
      avg.time /= (long)NUMTIMES;
      avg.millitm += i * 1000;
      avg.millitm /= NUMTIMES;

      sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
      avgFile.write(tmp, strlen(tmp));
//    sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
      logFile.write(tmp, strlen(tmp));

      for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
            {
            avg.time+=benchmarks[i][1].time;
            avg.millitm+=benchmarks[i][1].millitm;
            if (avg.millitm % 1000 != avg.millitm)
                  {
                  avg.time += (long)(avg.millitm / (short)1000);
                  avg.millitm %= (short)1000;
                  }
            }
      sprintf(tmp, "%s Tot. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
      logFile.write(tmp, strlen(tmp));
      i = (int) (avg.time % (long)NUMTIMES);
      avg.time /= (long)NUMTIMES;
      avg.millitm += i * 1000;
      avg.millitm /= NUMTIMES;
      sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
      avgFile.write(tmp, strlen(tmp));
//    sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
      logFile.write(tmp, strlen(tmp));

      for(i=0, avgTimeTaken=0; i<NUMTIMES; i++)
            avgTimeTaken+=timeTaken[i];
      avgTimeTaken = avgTimeTaken / NUMTIMES;
      sprintf(tmp, "%s Avg. Time Taken:%d\n", mapFileName, avgTimeTaken);
      avgFile.write(tmp, strlen(tmp));
      logFile.write(tmp, strlen(tmp));

      for(i=0, avgDist=0.0; i<NUMTIMES; i++)
            avgDist += distRobotTravelled[i];
      avgDist = avgDist / ((float)NUMTIMES);
      sprintf(tmp, "%s Avg. Dist Travelled:%f\n", mapFileName, avgDist);
      avgFile.write(tmp, strlen(tmp));
      logFile.write(tmp, strlen(tmp));

      sprintf(tmp, "%s ====================\n", mapFileName);
      avgFile.write(tmp, strlen(tmp));
//    sprintf(tmp, "%s ====================\n", mapFileName);
      logFile.write(tmp, strlen(tmp));
      }

// domain.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
```

```
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00

#define FROM_FRONT 0
#define FROM_BACK 1
#define FROM_LEFT 2
#define FROM_RIGHT 3
#define FROM_FRONTRIGHT 4
#define FROM_FRONTLEFT 5
#define FROM_BACKRIGHT 6
#define FROM_BACKLEFT 7
#define FROM_UP 8
#define FROM_DOWN 9
#define FROM_NOWHERE 10

#define NOCOST (float)0.0
#define NORMAL (float)1.0
#define NORMAL_DIAG (float)1.414214
#define BLOCKED (float)1000.0

typedef struct
       {
       float dist;
       int from;
       char type;
       float cost[9];
       } POINT;

class Domain
       {
       public:
              Domain(int, int, int, char*);
              ~Domain();
              void AdvanceTime(void);
      float CheckLine(int, int, int, int, int, int);
              void DrawDomain(void);
      int GetDomainLength(void) { return(domainLength); }
      int GetDomainTimeSlices(void) { return(domainTimeSlices); }
      int GetDomainWidth(void) { return(domainWidth); }
              POINT* GetPoint(int, int, int);
      float GetPointCost(int, int, int, int);
              char GetPointFrom(int, int, int);
              char GetPointType(int, int, int);
              int IsPointClear(int, int, int);
              int IsPointGoal(int, int, int);
              int IsPointNearObject(int, int, int);
              int IsPointObject(int, int, int);
              int IsPointStart(int, int, int);
      int IsPointVertex(int, int, int);
              float MoveRobot(int, int, int, int, int, int);
              void SetPointFrom(int, int, int, int);
              void SetPointType(int, int, int, char);
       private:
              void AgeTimeSlices(void);
      int CalcRobotDir(int, int, int, int, int, int);
              int ClearAdjPointOK(int, int, int);
              void ClearMobileObject(int, int, int);
              void ClearVerticesInTimeSlice(int);
              void DrawTimeSlice(int);
              void InitTimeSlice(int);
              void MarkMobileObject(int, int, int);
              void MoveMobileObject(OBJECT_NODE*);
              void MoveMobileObjects(void);
              void SetAdjObjsInTimeSlice(int);
              void SetGoalFromFile(char*);
              void SetMobileObjsFromFile(char*);
              void SetPermObjsInTimeSlice(int);
              void SetStartFromFile(char*);
              void SetVerticesInTimeSlice(int);

              ObjectList objList;
              POINT** domainHead;
      int domainWidth;
      int domainLength;
      int domainTimeSlices;
       };
```

```
// domain.cpp
#include <iostream.h>
#include <fstream.h>
#include <alloc.h>          // for coreleft()
#include <stdio.h>
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <conio.h>
#include "object.hpp"
#include "domain.hpp"

Domain::Domain(int numTimeSlices, int width, int length, char*mapFileName)
       {
       int i;

       cout << "Contructing Domain class\n";
       domainTimeSlices = numTimeSlices;
       domainWidth = width;
       domainLength = length;

       cout << "FreeHeap:" << farcoreleft() << "\n";
       cout << "Amount needed for one timeslice:";
       cout << width*length*sizeof(POINT) << "\n";

       domainHead = (POINT**)farcalloc(numTimeSlices, sizeof(POINT**));
       if (domainHead == (POINT**)NULL)
              {
              cout << "Not enough memory to build model of world\n";
              domainHead = (POINT***)NULL;
              domainTimeSlices = domainWidth = domainLength = 0;
              return;
              }

       for(i=0; i<numTimeSlices; i++)
              {
              domainHead[i] = (POINT*)farcalloc(width*length, sizeof(POINT));
              cout << "FreeHeap after timeslice allocated:" << farcoreleft() << "\n";
              if (domainHead[i] == (POINT*)NULL)
                     {
                     if (i==0)
                            cout << "Not enough memory to build model of world\n";
                     else
                            cout << "Only enough memory to build " << i << " of the ";
                            cout << numTimeSlices << " timeslices in model of world\n";
                     domainTimeSlices = i;
                     break;
                     }
              InitTimeSlice(i);
              SetPermObjsInTimeSlice(i);
              }
       SetStartFromFile(mapFileName);
       SetGoalFromFile(mapFileName);
       SetMobileObjsFromFile(mapFileName);
       for(i=0; i<domainTimeSlices; i++)
              {
              SetAdjObjsInTimeSlice(i);
              SetVerticesInTimeSlice(i);
              }
       for(i=0; i<domainTimeSlices-1; i++)
              AdvanceTime();
       cout << "FreeHeap after Domain allocated:" << farcoreleft() << "\n";
       }


Domain::~Domain()
       {
       POINT *tmp;
       int i;

       cout << "Destructing Domain class\n";
       if (domainHead == (POINT**)NULL)
              {
              cout << "Not freeing domain - DomainHead was NULL\n";
              return;
              }
       for(i=0; i<domainTimeSlices; i++)
              {
              tmp = domainHead[i];
              farfree(tmp);
              }
       farfree(domainHead);
       domainHead = (POINT**)NULL;
```

```
	cout << "FreeHeap after Domain deallocated:" << farcoreleft() << "\n";
//	getch();
	}


void Domain::AdvanceTime()
	{
	AgeTimeSlices();
//	DrawDomain();
	ClearVerticesInTimeSlice(domainTimeSlices-1);
	MoveMobileObjects();
//	DrawDomain();
	SetVerticesInTimeSlice(domainTimeSlices-1);
//	DrawDomain();
	}


////////////////////////////////////////
//
// Copy the contents of every timeslice
// into the previous timeslice
// [0] = [1], [1] = [2], etc.
// Leave the last timeslice unchanged.
// Another routine will decide the moves
// for all the mobile objects.
//
// A quick way to do this is moving ptrs
// to the timeslices and only copying the
// contents of the last timeslice over the
// contents of the first timeslice
//
///////////////////////////////////////
void Domain::AgeTimeSlices()
	{
	int i, j;
	POINT *tmp;

	tmp = domainHead[0];
	for(i=0, j=1; i<domainTimeSlices-1; i++, j++)
		domainHead[i] = domainHead[j];
	domainHead[domainTimeSlices-1] = tmp;
	_fmemcpy(domainHead[domainTimeSlices-1],
				domainHead[domainTimeSlices-2],
				sizeof(*domainHead[domainTimeSlices-1]));
	}


int Domain::CalcRobotDir(int st, int sw, int sl, int et, int ew, int el)
   {
	if ((!IsPointStart(st, sw, sl))
	|| (IsPointGoal(st, sw, sl)))
		return(clear);

	if ((ew - sw > 0)
	&& (el - sl > 0))
		{
		sw++, sl++;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(frontright);
		else
			return(clear);
		}

	if ((ew - sw > 0)
	&& (el - sl == 0))
		{
		sw++;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(right);
		else
			return(clear);
		}

	if ((ew - sw > 0)
	&& (el - sl < 0))
		{
		sw++, sl--;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
```

```
				return(backright);
			else
				return(clear);
			}

		if ((ew - sw == 0)
		&& (el - sl > 0))
			{
			sl++;
			if ((IsPointClear(st, sw, sl))
			|| (IsPointVertex(st, sw, sl))
			|| (IsPointGoal(st, sw, sl)))
				return(front);
			else
				return(clear);
			}

		if ((ew - sw == 0)  // silly, but just in case
		&& (el - sl == 0))
			{
			return(clear);
			}

		if ((ew - sw == 0)
		&& (el - sl < 0))
			{
			sl--;
			if ((IsPointClear(st, sw, sl))
			|| (IsPointVertex(st, sw, sl))
			|| (IsPointGoal(st, sw, sl)))
				return(back);
			else
				return(clear);
			}

		if ((ew - sw < 0)
		&& (el - sl > 0))
			{
			sw--, sl++;
			if ((IsPointClear(st, sw, sl))
			|| (IsPointVertex(st, sw, sl))
			|| (IsPointGoal(st, sw, sl)))
				return(frontleft);
			else
				return(clear);
			}

		if ((ew - sw < 0)
		&& (el - sl == 0))
			{
			sw--;
			if ((IsPointClear(st, sw, sl))
			|| (IsPointVertex(st, sw, sl))
			|| (IsPointGoal(st, sw, sl)))
				return(left);
			else
				return(clear);
			}

		if ((ew - sw < 0)
		&& (el - sl < 0))
			{
			sw--, sl--;
			if ((IsPointClear(st, sw, sl))
			|| (IsPointVertex(st, sw, sl))
			|| (IsPointGoal(st, sw, sl)))
				return(backleft);
			else
				return(clear);
			}
		return(clear);
		}


float Domain::CheckLine(int st, int sw, int sl, int et, int ew, int el)
		{
//		static int displayCounter=1;
		int tmpt, tmpw, tmpl;
		float dist;

		for(dist=0.0, tmpt=et-st, tmpw=ew-sw, tmpl=el-sl;
			(tmpt != 0) || (tmpw != 0) || (tmpl != 0);
			tmpt=et-st, tmpw=ew-sw, tmpl=el-sl)
```

```
        }
        if (tmpt < 0)
                {
                return(-1.0);
                }
        if ((tmpw > 0)
        && (tmpl > 0))
                {
                dist += GetPointCost(st, sw++, sl++, frontright);
                if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
 if ((tmpw > 0)
        && (tmpl == 0))
                {
    dist += GetPointCost(st, sw++, sl, right);
        if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
                }
        if ((tmpw > 0)
        && (tmpl < 0))
                {
    dist += GetPointCost(st, sw++, sl--, backright);
                if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
        if ((tmpw < 0)
        && (tmpl > 0))
                {
    dist += GetPointCost(st, sw--, sl++, frontleft);
        if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
        if ((tmpw < 0)
        && (tmpl == 0))
                {
    dist += GetPointCost(st, sw--, sl, left);
        if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
        if ((tmpw < 0)
        && (tmpl < 0))
                {
    dist += GetPointCost(st, sw--, sl--, backleft);
                if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
                }
        if ((tmpw == 0)
        && (tmpl > 0))
                {
    dist += GetPointCost(st, sw, sl++, front);
        if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
        if ((tmpw == 0)
        && (tmpl < 0))
                {
    dist += GetPointCost(st, sw, sl--, back);
        if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
if ((tmpw == 0)
&& (tmpl == 0)
&& (tmpt >0))
                {
    dist += GetPointCost(st++, sw, sl, up);
        if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
```

```
                        return(-1.0);
          continue;
          }
              }
//      cout << "CheckLine:" << displayCounter++ << ":returned" << dist << "\n";
        return(dist);
        }


int Domain::ClearAdjPointOK(int t, int w, int l)
        {
        char type;
        int a,b;

        for(a=w-1;a<=w+1;a++)
                {
                if ((a < 0) || (a >= domainWidth))
                        continue;
                for(b=l-1;b<=l+1;b++)
                        {
                        if ((b < 0) || (b >= domainLength))
                                continue;
                        if ((a == w) && (b == l))
                                continue;

          type = GetPointType(t, a, b);
                        if ((type == OBJECT)
                        ||  (type == MOBILE_OBJECT))
                                return(FALSE);
                        }
                }
        return(TRUE);
        }


void Domain::ClearMobileObject(int t, int w, int l)
   {
        int i, j;

        if (GetPointType(t, w, l) == MOBILE_OBJECT)
                {
                if (ClearAdjPointOK(t, w, l))
                        {
                        SetPointType(t, w, l, CLEAR);
                        SetPointFrom(t, w, l, FROM_NOWHERE);
                        }
                else
                        {
                        SetPointType(t, w, l, ADJ_TO_OBJECT);
                        SetPointFrom(t, w, l, FROM_NOWHERE);
                        }
                }
        for(i=w-1, j=l-1; j<l+2;)
                {
       if ((GetPointType(t, i, j) == ADJ_TO_OBJECT)
                && (ClearAdjPointOK(t, i, j)))
                {
                SetPointType(t, i, j, CLEAR);
                        SetPointFrom(t, i, j, FROM_NOWHERE);
                        }
                if (i == w+1)
                        {
                        i = w-1;
                        j++;
                        }
                else
                        i++;
                }
        }


void Domain::ClearVerticesInTimeSlice(int t)
        {
        int w, l;

        for(w=0; w<domainWidth; w++)
                {
                for(l=0; l<domainLength; l++)
                        {
                        if (GetPointType(t, w, l) == VERTEX)
            {
                                SetPointFrom(t, w, l, FROM_NOWHERE);
                                SetPointType(t, w, l, CLEAR);
            continue;
```

```
        {
                if (GetPointType(t, w, l) == MOBILE_OBJECT)
                        {
                ClearMobileObject(t, w, l);
                        }
                }
            }
        }

void Domain::DrawDomain()
        {
        int i;

        for (i=0; i< domainTimeSlices; i++)
                {
                DrawTimeSlice(i);
//              getch();
                }
        }

void Domain::DrawTimeSlice(int timeSlice)
        {
        char type;
        int a,b,y;

        clrscr();
        gotoxy(1,1);
        cout << "time=t+" << timeSlice;

        for(b=0, y=2;b<domainLength; b++, y++)
                {
                gotoxy(1, y);
                for(a=0; a<domainWidth; a++)
                        {
                        type = GetPointType(timeSlice, a, b);
                        switch(type)
                                {
                                case GOAL:
                                case START:
                                case ADJ_TO_OBJECT:
                                case OBJECT:
                                case MOBILE_OBJECT:
                                case VERTEX:
                                        cout << type;
                                        break;
                                case CLEAR:
                                        switch(GetPointFrom(timeSlice, a, b))
                                                {
                                                case FROM_RIGHT:
                                                        cout << "R";
                                                        break;
                                                case FROM_LEFT:
                                                        cout << "L";
                                                        break;
                                                case FROM_UP:
                                                        cout << "U";
                                                        break;
                                                case FROM_DOWN:
                                                        cout << "D";
                                                        break;
                                                case FROM_FRONT:
                                                        cout << "F";
                                                        break;
                                                case FROM_BACK:
                                                        cout << "B";
                                                        break;
                                                case FROM_BACKLEFT:
                                                        cout << "T";
                                                        break;
                                                case FROM_BACKRIGHT:
                                                        cout << "U";
                                                        break;
                                                case FROM_FRONTLEFT:
                                                        cout << "V";
                                                        break;
                                                case FROM_FRONTRIGHT:
                                                        cout << "W";
                                                        break;
                                                case FROM_NOWHERE:
                                                default:
                                                        cout << ".";
                                                        break;
```

```
                                                }
                                        break;
                                default:
                                        cout << "?";
                                        break;
                                }
                        }
                cout << "\n";
                }
        }


POINT* Domain::GetPoint(int timeSlice, int width, int length)
        {
    POINT *tmp, *tmp2;

    tmp = domainHead[timeSlice];
        tmp2 = tmp + (domainWidth*width) + length;
    return(tmp2);
    }


float Domain::GetPointCost(int timeSlice, int width, int length, int dir)
    {
    POINT *tmp;

    tmp = GetPoint(timeSlice, width, length);
        return(tmp->cost[dir]);
    }

char Domain::GetPointFrom(int timeSlice, int width, int length)
    {
    POINT *tmp;

    tmp = GetPoint(timeSlice, width, length);
        return(tmp->from);
    }


char Domain::GetPointType(int timeSlice, int width, int length)
        {
    POINT *tmp;

        tmp = GetPoint(timeSlice, width, length);
        return(tmp->type);
        }


void Domain::InitTimeSlice(int timeSlice)
        {
        POINT*tmp;
    int a,b;

    for(a=0;a<domainWidth;a++)
                {
                for(b=0;b<domainLength;b++)
                        {
         tmp = GetPoint(timeSlice, a, b);
                        tmp->dist=0.0;
                        tmp->from=FROM_NOWHERE;
                        tmp->type=CLEAR;

                        if (timeSlice == domainTimeSlices-1)
                                tmp->cost[FROM_UP] = BLOCKED;
                        else
                                tmp->cost[FROM_UP] = NORMAL;

                        if ((a == 0) && (b == 0))
                                {
                                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                                tmp->cost[FROM_RIGHT] = NORMAL;
                                tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                tmp->cost[FROM_BACK] = BLOCKED;
                                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                                tmp->cost[FROM_LEFT] = BLOCKED;
                                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                                tmp->cost[FROM_FRONT] = NORMAL;
                                continue;
                                }
                        if ((a == 0) && (b < domainLength-1))
                                {
                                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                                tmp->cost[FROM_RIGHT] = NORMAL;
                                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
```

```
            tmp->cost[FROM_BACK] = NORMAL;
            tmp->cost[FROM_BACKLEFT] = BLOCKED;
            tmp->cost[FROM_LEFT] = BLOCKED;
            tmp->cost[FROM_FRONTLEFT] = BLOCKED;
            tmp->cost[FROM_FRONT] = NORMAL;
            continue;
            }
    if ((a == 0) && (b==domainLength-1))
            {
            tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
            tmp->cost[FROM_RIGHT] = NORMAL;
            tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
            tmp->cost[FROM_BACK] = NORMAL;
            tmp->cost[FROM_BACKLEFT] = BLOCKED;
            tmp->cost[FROM_LEFT] = BLOCKED;
            tmp->cost[FROM_FRONTLEFT] = BLOCKED;
            tmp->cost[FROM_FRONT] = BLOCKED;
            continue;
            }
    if ((a < domainWidth-1) && (b == 0))
            {
            tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
            tmp->cost[FROM_RIGHT] = NORMAL;
            tmp->cost[FROM_BACKRIGHT] = BLOCKED;
            tmp->cost[FROM_BACK] = BLOCKED;
            tmp->cost[FROM_BACKLEFT] = BLOCKED;
            tmp->cost[FROM_LEFT] = NORMAL;
            tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
            tmp->cost[FROM_FRONT] = NORMAL;
            continue;
            }
    if ((a < domainLength-1) && (b < domainWidth-1))
            {
            tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
            tmp->cost[FROM_RIGHT] = NORMAL;
            tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
            tmp->cost[FROM_BACK] = NORMAL;
            tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
            tmp->cost[FROM_LEFT] = NORMAL;
            tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
            tmp->cost[FROM_FRONT] = NORMAL;
            continue;
            }
    if ((a < domainWidth-1) && (b == domainLength-1))
            {
            tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
            tmp->cost[FROM_RIGHT] = NORMAL;
            tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
            tmp->cost[FROM_BACK] = NORMAL;
            tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
            tmp->cost[FROM_LEFT] = NORMAL;
            tmp->cost[FROM_FRONTLEFT] = BLOCKED;
            tmp->cost[FROM_FRONT] = BLOCKED;
            continue;
            }
    if ((a == domainWidth-1) && (b == 0))
            {
            tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
            tmp->cost[FROM_RIGHT] = BLOCKED;
            tmp->cost[FROM_BACKRIGHT] = BLOCKED;
            tmp->cost[FROM_BACK] = BLOCKED;
            tmp->cost[FROM_BACKLEFT] = BLOCKED;
            tmp->cost[FROM_LEFT] = NORMAL;
            tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
            tmp->cost[FROM_FRONT] = NORMAL;
            continue;
            }
    if ((a == domainWidth-1) && (b < domainWidth-1))
            {
            tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
            tmp->cost[FROM_RIGHT] = BLOCKED;
            tmp->cost[FROM_BACKRIGHT] = BLOCKED;
            tmp->cost[FROM_BACK] = NORMAL;
            tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
            tmp->cost[FROM_LEFT] = NORMAL;
            tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
            tmp->cost[FROM_FRONT] = NORMAL;
            continue;
            }
    if ((a == domainLength-1) && (b == domainWidth-1))
            {
            tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
            tmp->cost[FROM_RIGHT] = BLOCKED;
            tmp->cost[FROM_BACKRIGHT] = BLOCKED;
```

```
                                tmp->cost[FROM_BACK] = NORMAL;
                                tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                                tmp->cost[FROM_LEFT] = NORMAL;
                                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                                tmp->cost[FROM_FRONT] = BLOCKED;
                                continue;
                                }
                        }
                }
        }


int Domain::IsPointClear(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case CLEAR:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointGoal(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case GOAL:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }



int Domain::IsPointNearObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
                case OBJECT:
                        return(OBJECT);
                case MOBILE_OBJECT:
                        return(MOBILE_OBJECT);
                case ADJ_TO_OBJECT:
                        return(ADJ_TO_OBJECT);
                default:
                        return(FALSE);
                }
        }

int Domain::IsPointObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
```

```
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
                case OBJECT:
                        return(OBJECT);
                case MOBILE_OBJECT:
                        return(MOBILE_OBJECT);
                case ADJ_TO_OBJECT:
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointStart(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case START:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointVertex(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case VERTEX:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


void Domain::MarkMobileObject(int t, int w, int l)
        {
        int i, j;

        if ((GetPointType(t, w, l) == CLEAR)
        || (GetPointType(t, w, l) == ADJ_TO_OBJECT))
                {
                SetPointType(t, w, l, MOBILE_OBJECT);
                SetPointFrom(t, w, l, FROM_NOWHERE);
                }
        for(i=w-1, j=l-1; j<l+2;)
                {
        if (GetPointType(t, i, j) == CLEAR)
                {
                SetPointType(t, i, j, ADJ_TO_OBJECT);
                SetPointFrom(t, i, j, FROM_NOWHERE);
                }
        if (i == w+1)
           {
           i = w-1;
           j++;
           }
        else
                        i++;
                }
        }


void Domain::MoveMobileObject(OBJECT_NODE*object)
        {
        int t=domainTimeSlices-1, w=(*object).w, l=(*object).l;
```

```
        ClearMobileObject(t, w, l);
        switch((*object).direction)
                {
                case front:
                        if (IsPointObject(t, w, l+1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).l++;
                        break;
                case frontleft:
                        if (IsPointObject(t, w-1, l+1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).l++, (*object).w--;
                        break;
                case left:
                        if (IsPointObject(t, w-1, l))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w--;
                        break;
                case backleft:
                        if (IsPointObject(t, w-1, l-1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w--, (*object).l--;
                        break;
                case back:
                        if (IsPointObject(t, w, l-1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).l--;
                        break;
                case backright:
                        if (IsPointObject(t, w+1, l-1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w++, (*object).l--;
                        break;
                case right:
                        if (IsPointObject(t, w+1, l))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w++;
                        break;
                case frontright:
                        if (IsPointObject(t, w+1, l+1))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w++, (*object).l++;
                        break;
                default:
                        cout << "*** unknown direction for mobile object ignored ***\n";
                        break;
                }
        MarkMobileObject(t, object->w, object->l);
        }


void Domain::MoveMobileObjects()
        {
        OBJECT_NODE*cur, *orig;

        orig = cur = objList.GetNextObject();
        if (cur == (OBJECT_NODE*)NULL)
        return;

        do
                {
                if ((*cur).velocity > 0)
                        MoveMobileObject(cur);

        cur = objList.GetNextObject();
                } while (cur != orig);
        }


float Domain::MoveRobot(int st, int sw, int sl, int et, int ew, int el)
        {
        float dist = 0.0;
        int dir;
```

```
if (!IsPointStart(st, sw, sl))
        return(dist);

dir = CalcRobotDir(st, sw, sl, et, ew, el);
SetPointType(st, sw, sl, CLEAR);
SetPointFrom(st, sw, sl, FROM_NOWHERE);

switch(dir)
        {
        case frontright:
                dist = GetPointCost(st, sw, sl, frontright);
                st++, sw++, sl++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sw--, sl--;
                        dist = 0.0;
                        }
                break;
        case right:
                dist = GetPointCost(st, sw, sl, right);
                st++, sw++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sw--;
                        dist = 0.0;
                        }
                break;
        case backright:
                dist = GetPointCost(st, sw, sl, backright);
                st++, sw++, sl--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sw--, sl++;
                        dist = 0.0;
                        }
                break;
        case frontleft:
                dist = GetPointCost(st, sw, sl, frontleft);
                st++, sw--, sl++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sw++, sl--;
                        dist = 0.0;
                        }
                break;
        case left:
                dist = GetPointCost(st, sw, sl, left);
                st++, sw--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sw++;
                        dist = 0.0;
                        }
                break;
        case backleft:
                dist = GetPointCost(st, sw, sl, backleft);
                st++, sw--, sl--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sw++, sl++;
                        dist = 0.0;
                        }
                break;
        case front:
                dist = GetPointCost(st, sw, sl, front);
                st++, sl++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sl--;
                        dist = 0.0;
```

```
                }
            break;
        case back:
            dist = GetPointCost(st, sw, sl, back);
            st++, sl--;
            if ((!IsPointClear(st, sw, sl))
            && (!IsPointVertex(st, sw, sl))
            && (!IsPointGoal(st, sw, sl)))
                {
                st--, sl++;
                dist = 0.0;
                }
            break;
        case up:
            dist = GetPointCost(st, sw, sl, up);
            st++;
            if ((!IsPointClear(st, sw, sl))
            && (!IsPointVertex(st, sw, sl))
            && (!IsPointGoal(st, sw, sl)))
                {
                st--;
                dist = 0.0;
                }
            break;
        default:
            break;
        }
    SetPointType(st, sw, sl, START);
    SetPointFrom(st, sw, sl, FROM_NOWHERE);
    return(dist);
    }


void Domain::SetAdjObjsInTimeSlice(int timeSlice)
    {
    int a,b;
    int w,l;

    for(w=0; w<domainWidth; w++)
        {
        for(l=0; l<domainLength; l++)
            {
            if (!IsPointObject(timeSlice, w, l))
                continue;

            for(a=w-1; a<=w+1; a++)
                {
                if ((a < 0) || (a >= domainWidth))
                    continue;
                for(b=l-1; b<=l+1; b++)
                    {
                    if ((b < 0) || (b >= domainLength))
                        continue;

                    if (GetPointType(timeSlice, a, b) == CLEAR)
                        {
                        SetPointFrom(timeSlice, a, b, FROM_NOWHERE);
                        SetPointType(timeSlice, a, b, ADJ_TO_OBJECT);
                        }
                    }
                }
            }
        }
    }


void Domain::SetGoalFromFile(char*fileName)
    {
    int i, w, l;
    char recType;
    char tmp[256];
    fstream dataFile;

    dataFile.open(fileName, ios::in);
    if (!dataFile)
        cout << "Unable to open " << fileName << ". No Goal added.\n";

    for(;;)
        {
        dataFile. getline(tmp, sizeof(tmp));
        if (strlen(tmp) == 0)
            break;
        if (tmp[0] == GOAL)
            {
```

```
//                  cout << "The GOAL line is: " << tmp << "\n";
                    if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                        {
            for(i=0; i<domainTimeSlices; i++)
               {
                        SetPointFrom(i, w, l, FROM_NOWHERE);
                        SetPointType(i, w, l, GOAL);
               }
                        break;
                        }
                    else
                        cout << "Improperly formatted line ignored\n";
                    }
             }
        dataFile.close();
        }


void Domain::SetMobileObjsFromFile(char*fileName)
        {
        int w, l;
        char recType;
        char tmp[256];
        fstream dataFile;
        OBJECT_NODE* newObj;

        dataFile.open(fileName, ios::in);
        if (!dataFile)
                cout << "Unable to open " << fileName << ". No Mobile objs added.\n";

        for(;;)
                {
                int i;

                dataFile. getline(tmp, sizeof(tmp));
                i = strlen(tmp);
                if (i <= 0)
                        break;
                cout << "Length of input line is: " << i << "\n";
                if ((tmp[0] != GOAL)
                && (tmp[0] != START))
                        {
//                      cout << "OBJECT line is: " << tmp << "\n";
                        if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                                {
                                SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
                                SetPointType(domainTimeSlices-1, w, l, MOBILE_OBJECT);
                                newObj = objList.BuildNewObject(w, l);
                                if (newObj)
                                        {
                                        objList.InsertNewObject(newObj);
//                                      cout << "Added obj to obj list\n";
                                        }
                                }
                        else
                                cout << "Improperly formatted line ignored\n";
                        }
                }
        dataFile.close();
        }


void Domain::SetPermObjsInTimeSlice(int timeSlice)
        {
        int w, l;

        for(w=0; w<domainWidth-5; w++)
                {
                l=2;
                switch(w)
                        {
                        case 3:
                        case 4:
                        case 5:
                                break;
                        default:
                                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                                SetPointType(timeSlice, w, l, OBJECT);
//                              SetAdjObjsInTimeSlice(timeSlice, w, l);
                                break;
                        }
                }
        for(w=0; w<domainWidth-10; w++)
                {
```

```
            l=7;
            SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
            SetPointType(timeSlice, w, l, OBJECT);
//          SetAdjObjsInTimeSlice(timeSlice, w, l);
            }
        for(l=0, w=domainWidth-10; l<domainLength-2; l++)
            {
            switch(l)
                {
                case 3:
                case 4:
                case 5:
                case 6:
                case 14:
                case 15:
                case 16:
                    break;
                default:
                    SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                    SetPointType(timeSlice, w, l, OBJECT);
//                  SetAdjObjsInTimeSlice(timeSlice, w, l);
                    break;
                }
            }
        for(l=2, w=domainWidth-5; l<domainLength-7; l++)
            {
            switch(l)
                {
                case 9:
                case 10:
                case 11:
                    break;
                default:
                    SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                    SetPointType(timeSlice, w, l, OBJECT);
//                  SetAdjObjsInTimeSlice(timeSlice, w, l);
                    break;
                }
            }
        for(w=domainWidth-5, l=domainLength-7; w<domainWidth; w++)
            {
            SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
            SetPointType(timeSlice, w, l, OBJECT);
//          SetAdjObjsInTimeSlice(timeSlice, w, l);
            }
        for(w=domainWidth-10, l=domainLength-2; w<domainWidth; w++)
            {
            SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
            SetPointType(timeSlice, w, l, OBJECT);
//          SetAdjObjsInTimeSlice(timeSlice, w, l);
            }
        }


void Domain::SetPointFrom(int timeSlice, int width, int length, int from)
        {
        POINT *tmp;

        if ((width < 0)
        || (length < 0)
        || (timeSlice < 0)
        || (width >= domainWidth)
        || (length >= domainLength)
        || (timeSlice >= domainTimeSlices))
            return;
        tmp = GetPoint(timeSlice, width, length);
        tmp->from = from;
        }

void Domain::SetPointType(int timeSlice, int width, int length, char type)
        {
        char oldType;
        POINT *tmp;

        if ((width < 0)
        || (length < 0)
        || (timeSlice < 0)
        || (width >= domainWidth)
        || (length >= domainLength)
        || (timeSlice >= domainTimeSlices))
            return;
        tmp = GetPoint(timeSlice, width, length);
        oldType = tmp->type;
        tmp->type = type;
```

```
	if (oldType != OBJECT)
		{
//		SetAdjObjsInTimeSlice(timeSlice, width, length);
		}
	}

void Domain::SetStartFromFile(char*fileName)
	{
	int w, l;
	char recType;
	char tmp[256];
	fstream dataFile;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Start added.\n";

	for(;;)
		{
		dataFile. getline(tmp, sizeof(tmp));
		if (strlen(tmp) == 0)
			break;
		if (tmp[0] == START)
			{
//			cout << "The START line is: " << tmp << "\n";
			if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
				{
				SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
				SetPointType(domainTimeSlices-1, w, l, START);
				break;
				}
			else
				cout << "Improperly formatted line ignored\n";
			}
		}
	dataFile.close();
	}

void Domain::SetVerticesInTimeSlice(int t)
	{
	int w, l;
	int diff_counter, corner_counter;
	int side_bits, corner_bits;

	for(w=0; w<domainWidth; w++)
		{
		for(l=0; l<domainLength; l++)
			{
			if (!IsPointClear(t, w, l))
				continue;

			corner_bits = BITMASK_CLEAR;
			corner_counter = 0;
			if (IsPointNearObject(t, w-1, l-1))
				{
				corner_bits |= BITMASK_LEFT;
				corner_bits |= BITMASK_TOP;
				corner_counter++;
				}
			if (IsPointNearObject(t, w-1, l+1))
				{
				corner_bits |= BITMASK_LEFT;
				corner_bits |= BITMASK_BOTTOM;
				corner_counter++;
				}
			if (IsPointNearObject(t, w+1, l+1))
				{
				corner_bits |= BITMASK_RIGHT;
				corner_bits |= BITMASK_BOTTOM;
				corner_counter++;
				}
			if (IsPointNearObject(t, w+1, l-1))
				{
				corner_bits |= BITMASK_RIGHT;
				corner_bits |= BITMASK_TOP;
				corner_counter++;
				}
			if (corner_bits != BITMASK_CLEAR)
				{
				side_bits = BITMASK_CLEAR;
				if (IsPointNearObject(t, w-1, l))
					side_bits |= BITMASK_LEFT;
```

```
                    if (IsPointNearObject(t, w, l-1))
                          side_bits |= BITMASK_TOP;
                    if (IsPointNearObject(t, w+1, l))
                          side_bits |= BITMASK_RIGHT;
                    if (IsPointNearObject(t, w, l+1))
                          side_bits |= BITMASK_BOTTOM;

                    for(diff_counter=0;
                          (side_bits != BITMASK_CLEAR)
                          || (corner_bits != BITMASK_CLEAR);
                          side_bits >>= 1, corner_bits >>= 1)
                          {
                          if ((corner_bits & (int)0x01)
                          && ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
                                {
                                diff_counter++;
                                continue;
                                }
                          if ((side_bits & (int)0x01)
                          && ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
                                {
                                diff_counter++;
                                continue;
                                }
                          }
                    if ((diff_counter > 2)
                    || ((diff_counter == 2) && (corner_counter == 1)))
                          {
                          SetPointFrom(t, w, l, FROM_NOWHERE);
                          SetPointType(t, w, l, VERTEX);
                          }
                    }
              }
        }
  }

// edge.hpp
typedef struct van
   {
      int t,w,l;
      float dist;
      struct van *prev, *next;
      } EDGE_NODE;

class EdgeList
      {
      public:
            EdgeList();
            ~EdgeList();
      EDGE_NODE* BuildNewEdge(int, int, int, float);
      void DelEdge(EDGE_NODE*);
            void DelEdgeToVertex(int, int, int);
      EDGE_NODE* GetFirstEdge(void);
      EDGE_NODE* GetNextEdge(EDGE_NODE*);
      void InsertNewEdge(EDGE_NODE*);
      void ListAllEdges(void);
   private:
      EDGE_NODE *edgeHead;
      };

// edge.cpp
#include <iostream.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include "edge.hpp"


EdgeList::EdgeList()
      {
      edgeHead = (EDGE_NODE*)NULL;
//    cout << "EdgeList Constructor\n";
      }


EdgeList::~EdgeList()
      {
      while (edgeHead != (EDGE_NODE*)NULL)
            DelEdge(edgeHead);
//    cout << "EdgeList Destructor\n";
      }


EDGE_NODE* EdgeList::BuildNewEdge(int t, int w, int l, float dist)
```

```
        {
        EDGE_NODE *a;

//      if (otherVertex != (void*)NULL)
//              {
                a = (EDGE_NODE*)malloc(sizeof(EDGE_NODE));
                if (a == (EDGE_NODE*)NULL)
                        {
                        cout << "Out of Memory in BuildNewEdge()\n";
                        exit(0);
                        }
                (*a).t = t;
                (*a).w = w;
                (*a).l = l;
                (*a).dist = dist;
                (*a).prev = (*a).next = (EDGE_NODE*)NULL;
                return(a);
//              }
//      return((EDGE_NODE*)NULL);
        }


void EdgeList::DelEdge(EDGE_NODE*a)
   {
   EDGE_NODE*tmp;

//      cout << "edge 0\n";
        if (a != (EDGE_NODE*)NULL)
                {
                if ((a->prev == (EDGE_NODE*)NULL)  // del last remaining edge
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 1\n";
                        free(a);
                        edgeHead = (EDGE_NODE*)NULL;
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge in middle
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 2\n";
                        tmp = a->prev;
                        tmp->next = a->next;
                        tmp = a->next;
                        tmp->prev = a->prev;
                        free(a);
                        return;
                        }
                if ((a->prev == (EDGE_NODE*)NULL)  // del edge at sol
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 3\n";
                        tmp = a->next;
                        tmp->prev = (EDGE_NODE*)NULL;
                        edgeHead = tmp;
                        free(a);
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge at eol
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 4\n";
                        tmp = a->prev;
                        tmp->next = (EDGE_NODE*)NULL;
                        free(a);
                        return;
                        }
                }
        }


void EdgeList::DelEdgeToVertex(int t, int w, int l)
        {
        EDGE_NODE*tmp;

        for(tmp = GetFirstEdge(); tmp != (EDGE_NODE*)NULL; tmp = GetNextEdge(tmp))
                {
                if ((tmp->t == t)
        && (tmp->w == w)
        && (tmp->l == l))
                        {
                        DelEdge(tmp);
                        break;
                        }
```

```
                {
        }

EDGE_NODE* EdgeList::GetFirstEdge()
        {
   return(edgeHead);
   }


EDGE_NODE* EdgeList::GetNextEdge(EDGE_NODE*cur)
   {
   return(cur->next);
   }


void EdgeList::InsertNewEdge(EDGE_NODE *edge)
        {
        EDGE_NODE*cur;

        if (edge == (EDGE_NODE*)NULL)
                return;

        if (edgeHead == (EDGE_NODE*)NULL)
                {
                edgeHead = edge;
                edge->prev = edge->next = (EDGE_NODE*)NULL;
//              cout << "Inserted edge into empty list ";
//              cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//              cout << "Dist = " << edge->dist << "\n";
                return;
                }

        for(cur=edgeHead; cur != (EDGE_NODE*)NULL; cur=(*cur).next)
                {
                if ( (edge->t > cur->t)
                || ((edge->t == cur->t)
                        && (edge->w > cur->w))
                || ((edge->t == cur->t)
                        && (edge->w == cur->w)
                        && (edge->l > cur->l)) )
                        {                               // insert after cur
                        if (cur->next == (EDGE_NODE*)NULL)
                                {
                                edge->next = (EDGE_NODE*)NULL;// no more so append to eol
                                edge->prev = cur;
                                cur->next = edge;
                                break;
                                }
                        else
                                continue;                       // try next one
                        }
                if ( (edge->t < cur->t)
                || ((edge->t == cur->t)
                        && (edge->w < cur->w))
                || ((edge->t < cur->t)
                        && (edge->w == cur->w)
                        && (edge->l < cur->l)) )
                        {                               // insert before cur
                        if (cur->prev == (EDGE_NODE*)NULL)
                                {
                                edge->prev = (EDGE_NODE*)NULL;  // at start of list
                                edge->next = cur;
                                cur->prev = edge;
                                edgeHead = edge;
                                break;
                                }
                        else
                                {
                                edge->prev = cur->prev;     // in middle/end of list
                                edge->next = cur;
                                cur->prev = edge;
                                cur = edge->prev;
                                cur->next = edge;
                                break;
                                }
                        }
                if ( (edge->t == cur->t)
                && (edge->w == cur->w)
                && (edge->l == cur->l) )
                        {
                        /* already here - replace it ! */
                        cout << "Already here - replacing values and disposing of new edge ";
        cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
```

```
	cout << "Dist = " << edge->dist << "\n";
			cur->w = edge->w;
			cur->l = edge->l;
			cur->dist = edge->dist;
			free(edge);
			break;
			}
		}
//	cout << "Inserted edge into list ";
//	cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//	cout << "Dist = " << edge->dist << "\n";
	}



void EdgeList::ListAllEdges()
	{
	EDGE_NODE *edge;

	for(edge = edgeHead; edge != (EDGE_NODE*)NULL; edge = (*edge).next)
		{
		cout << "Edge to (";
		cout << (*edge).t << ",";
		cout << (*edge).w << ",";
		cout << (*edge).l << "). Dist is " << (*edge).dist << "\n";
		}
	cout << "-----------------\n";
	}

// object.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define NUM_DIRS 7 /* this is the 8 horizontal directions; (0->7) */
enum directions { front, back, left, right, frontright, frontleft, backright, backleft, up, down, clear};


typedef struct o
	{
	int direction, velocity;
	int w, l;
	struct o *prev, *next;
	} OBJECT_NODE;


class ObjectList
	{
	public:
		ObjectList();
		~ObjectList();
		OBJECT_NODE* BuildNewObject(int, int);
	void DelAllObjects(void);
	void DelObject(OBJECT_NODE*);
	OBJECT_NODE* GetNextObject(void);
		void InsertNewObject(OBJECT_NODE*);
	void ListAllObjects(void);
	private:
		OBJECT_NODE *objectHead;
	};

// object.cpp
#include <iostream.h>
#include <alloc.h>		// for coreleft()
#include <stdlib.h>		// for itoa()
#include <string.h>		// for strcpy()
// #include "edge.hpp"
#include "object.hpp"

ObjectList::ObjectList()
	{
	objectHead = (OBJECT_NODE*)NULL;
	cout << "ObjectList Constructor\n";
	}
```

```
ObjectList::~ObjectList()
        {
        cout << "ObjectList Destructor\n";
        DelAllObjects();
        }


OBJECT_NODE* ObjectList::BuildNewObject(int w, int l)
        {
        OBJECT_NODE *newPtr;

        newPtr = (OBJECT_NODE*)malloc(sizeof(OBJECT_NODE));
        if (newPtr == (OBJECT_NODE*)NULL)
                {
                cout << "Out of memory in BuildNewObject()\n";
                return(NULL);
                }
        (*newPtr).direction = rand() % NUM_DIRS;
        (*newPtr).velocity = 1;
        (*newPtr).prev = (*newPtr).next = (OBJECT_NODE*)NULL;
        (*newPtr).w = w;
        (*newPtr).l = l;
        return(newPtr);
        }


void ObjectList::DelAllObjects()
        {
        OBJECT_NODE *tmp;

        while (objectHead != (OBJECT_NODE*)NULL)
                {
                tmp = objectHead;
                objectHead = (*objectHead).next;
                DelObject(tmp);
                }
        }


void ObjectList::DelObject(OBJECT_NODE *todie)
    {
        OBJECT_NODE *cur;

        if (((*todie).prev != (OBJECT_NODE*)NULL)
        && ((*todie).next != (OBJECT_NODE*)NULL))
                {
//              cout << "Deleted Object (" << todie->w << "," << todie->l << ")\n";
                cur=(*todie).prev;
                if (cur == todie)
                        {
                        objectHead = (OBJECT_NODE*)NULL;
//                      cout << "Object List Empty\n";
                        free(todie);
                        return;
                        }
                else
                        {
                        (*cur).next=(*todie).next;
                        cur = (*todie).next;
                        (*cur).prev = (*todie).prev;
                        if (objectHead == todie)
                                objectHead = (*todie).next;
        free(todie);
          return;
                        }
                }
        cout << "***Did NOT delete rotton Object (" << todie->w << "," << todie->l << ")\n";
        }


OBJECT_NODE* ObjectList::GetNextObject()
   {
   OBJECT_NODE*tmp;

        if (objectHead == (OBJECT_NODE*)NULL)
                return((OBJECT_NODE*)NULL);

        tmp = objectHead;
   objectHead = objectHead->next;
   return(tmp);
   }
```

```
void ObjectList::InsertNewObject(OBJECT_NODE*newPtr)
        {
        OBJECT_NODE *cur;

        if (newPtr == (OBJECT_NODE*)NULL)
                return;

        if (objectHead == (OBJECT_NODE*)NULL)
                {
                objectHead = newPtr;
                (*newPtr).prev = (*newPtr).next = newPtr;
                }
        else
                {
                /* insert before first node */
                cur = objectHead;
                (*newPtr).next = cur;
                (*newPtr).prev = (*cur).prev;
                (*cur).prev = newPtr;

                cur = (*newPtr).prev;
                (*cur).next = newPtr;
                objectHead = newPtr;
        }
//      cout << "Inserted object (" << newPtr->w << "," << newPtr->l << ")\n";
        }


void ObjectList::ListAllObjects()
        {
   OBJECT_NODE*tmp;

   if (objectHead == (OBJECT_NODE*)NULL)
      {
      cout << "ObjectList is empty\n";
      return;
      }
   for(tmp=objectHead;;tmp = tmp->next)
      {
       cout << "Object:(" << tmp->w << "," << tmp->l << ")\n";
      tmp = tmp->next;
      if (tmp == objectHead)
         {
         cout << "ObjectList ended\n";
         break;
         }
      }
       }

// vertex.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00


typedef struct vn
        {
        int t,w,l;
        char nodeType;
        struct vn *searchPrev, *searchNext;
        float searchDist;
        int searchMarker;
        EdgeList *edgeList;
        struct vn *pathFrom, *pathTo;
        struct vn *prev, *next;
        } VERTEX_NODE;
```

```
class VertexList
	{
	public:
		VertexList();
		~VertexList();
		void AddToSearchList(VERTEX_NODE*);
		VERTEX_NODE* BuildNewVertex(int,int,int,char);
		int CalcRobotDir(VERTEX_NODE*);
		void DelAllVertices(void);
	void DelVertex(VERTEX_NODE*);
	int FindPath(void);
	VERTEX_NODE* FindVertex(int, int, int);
		VERTEX_NODE* GetFirstVertex(void);
	VERTEX_NODE* GetGoalVertex(void);
		VERTEX_NODE* GetNextVertex(VERTEX_NODE*);
	VERTEX_NODE* GetStartVertex(void);
	void InsertAllVertices(void);
		void InsertNewVertex(VERTEX_NODE*);
	void ListAllVertices(void);
	void ListSearchList(void);
//		void MoveRobot(int, int, int, int);
		void MarkPath(VERTEX_NODE*);
		void RemoveFromSearchList(VERTEX_NODE*);
		void TrimSearchList(void);
	private:
		VERTEX_NODE* vertexHead;
	VERTEX_NODE* searchHead;
	int searchMarker;
	float searchTrimDist;
	};

// vertex.cpp
#include <iostream.h>
#include <conio.h>		// for getch()
#include <alloc.h>		// for coreleft()
#include <stdlib.h>		// for itoa()
#include <string.h>		// for strcpy()
#include "edge.hpp"
#include "vertex.hpp"

VertexList::VertexList()
	{
	vertexHead = (VERTEX_NODE*)NULL;
   searchHead = (VERTEX_NODE*)NULL;
   searchMarker = 0;
   searchTrimDist = -1.0;
   cout << "Initialised Vertex class\n";
	}



VertexList::~VertexList()
	{
	cout << "VertexList destructor started\n";
	DelAllVertices();
	cout << "VertexList destructor ended!\n";
	}


void VertexList::AddToSearchList(VERTEX_NODE*a)
	{
	VERTEX_NODE *tmp;

	if (a == (VERTEX_NODE*)NULL)
		return;

	if (((*a).searchPrev != (VERTEX_NODE*)NULL) // if already in fringe
	|| ((*a).searchNext != (VERTEX_NODE*)NULL)) // list, dont add again
		return;

	if (searchHead == (VERTEX_NODE*)NULL)
		{
		searchHead = a;
		a->pathFrom = a->pathTo = (VERTEX_NODE*)NULL;
		a->searchPrev = a->searchNext = a;
		return;
		}

   for (tmp = searchHead;tmp != (VERTEX_NODE*)NULL;)
      {
		if (a->searchDist <= tmp->searchDist)
			{
			a->searchPrev = tmp->searchPrev;
			a->searchNext = tmp;
```

```
                        tmp->searchPrev = a;
                        tmp = a->searchPrev;
                        tmp->searchNext = a;
                        if (searchHead == a->searchNext)
                              searchHead = a;
                        return;
                        }
                  if (tmp->searchNext == searchHead)
                        {
                        a->searchPrev = tmp;
                        a->searchNext = searchHead;
                        tmp->searchNext = searchHead->searchPrev = a;
                        return;
                        }
                  else
                        tmp = tmp->searchNext;
                  }
            }


VERTEX_NODE* VertexList::BuildNewVertex(int t, int w, int l, char nodeType)
            {
            VERTEX_NODE *newPtr;

            newPtr = (VERTEX_NODE*)malloc(sizeof(VERTEX_NODE));
            if (newPtr == (VERTEX_NODE*)NULL)
                  {
                  cout << "Malloc() failed in BuildNewVertex!\n";
                  exit(0);
                  }
            (*newPtr).t = t;
            (*newPtr).w = w;
            (*newPtr).l = l;
            (*newPtr).nodeType = nodeType;
            (*newPtr).edgeList = new EdgeList();
            (*newPtr).pathTo = (*newPtr).pathFrom = (VERTEX_NODE*)NULL;
            (*newPtr).searchDist = 0.0;
            (*newPtr).searchPrev = (*newPtr).searchNext = (VERTEX_NODE*)NULL;
            (*newPtr).prev = (*newPtr).next = (VERTEX_NODE*)NULL;
            if (nodeType == START)
                  (*newPtr).searchMarker = searchMarker+1;
            else
                  (*newPtr).searchMarker = searchMarker;
            return(newPtr);
            }


void VertexList::DelAllVertices()
   {
            while (vertexHead != (VERTEX_NODE*)NULL)
                  {
//                cout << "About to delete (" << vertexHead->t << "," << vertexHead->w ;
//                cout << "," << vertexHead->l << ") at addr: " << vertexHead ;
//                cout << ": Prev:" << vertexHead->prev << ": Next:" << vertexHead->next <<"\n";
                  DelVertex(vertexHead);
                  }
            cout << "----------\n";
//          getch();
            }


void VertexList::DelVertex(VERTEX_NODE *todie)
            {
            VERTEX_NODE *tmp;
            EDGE_NODE*a;

            if (todie == (VERTEX_NODE*)NULL)
                  return;

            for(a = todie->edgeList->GetFirstEdge();
                  a != (EDGE_NODE*)NULL;
                  a = todie->edgeList->GetFirstEdge())
                  {
          todie->edgeList->DelEdgeToVertex(a->t, a->w, a->l);
          tmp = FindVertex(a->t, a->w, a->l);
          if (tmp != (VERTEX_NODE*)NULL)
             {
             tmp->edgeList->DelEdgeToVertex(todie->t, todie->w, todie->l);
             }
//                tmp = (VERTEX_NODE*)(a->otherVertex);
//                cout << "On vertex (" << todie->t << "," << todie->w << ",";
//                cout << todie->l << "):Del edge to (" << tmp->t << "," << tmp->w << ",";
```

```
//              cout << tmp->l << ")\n";
//              cout << "todie at:" << todie << ", a at:" << a << "\n";
//              cout << "  tmp at:" << tmp << "\n";
//              todie->edgeList->DelEdgeToVertex((void*)tmp);
//              cout << "On vertex (" << tmp->t << "," << tmp->w << ",";
//              cout << tmp->l << "):Del edge to (" << todie->t << "," << todie->w << ",";
//              cout << todie->l << ")\n";
//              tmp->edgeList->DelEdgeToVertex((void*)todie);
                }
        delete todie->edgeList;
        RemoveFromSearchList(todie);

        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
                {
                vertexHead = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
                tmp = (*todie).prev;
                (*tmp).next = (*todie).next;
                tmp = (*todie).next;
                (*tmp).prev = (*todie).prev;
                free(todie);
                return;
                }
        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
                vertexHead = tmp = (*todie).next;
                (*tmp).prev = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
                {
                tmp = (*todie).prev;
                (*tmp).next = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        }

int VertexList::FindPath()
    {
        EDGE_NODE *e;
        int goalFound=FALSE;
        float dist;
        VERTEX_NODE *cur, *adj, *dest;

        if (searchHead->nodeType == START)
                searchMarker = searchHead->searchMarker;
        else
                return(FALSE);

    dest = GetGoalVertex();
    if (dest == (VERTEX_NODE*)NULL)
        return(FALSE);

        for (cur = searchHead; cur != (VERTEX_NODE*)NULL; cur = searchHead)
                {
                if ((cur->searchDist >= searchTrimDist)
                && (searchTrimDist > 0.0))
                        {
                        RemoveFromSearchList(cur);
                        continue;
                        }
                for(e = cur->edgeList->GetFirstEdge();
                        e != (EDGE_NODE*)NULL;
                        e = cur->edgeList->GetNextEdge(e))
                        {
          adj = FindVertex(e->t, e->w, e->l);
          if (adj == (VERTEX_NODE*)NULL)
             continue;

//         adj = (VERTEX_NODE*) e->otherVertex;
                        dist = cur->searchDist + e->dist;

          if ( (adj->searchMarker != searchMarker)
```

```
                    || ((adj->searchMarker == searchMarker)
                        && (adj->searchDist > dist)) )
                        {
                        if ((dist < searchTrimDist)
                        || (searchTrimDist <= 0.0))
                            {
                            adj->pathFrom = cur;
                            adj->searchMarker = searchMarker;
                            adj->searchDist = dist;
                            AddToSearchList(adj);
                            if (adj->nodeType == GOAL)
                                {
                                goalFound = TRUE;
                                searchTrimDist = dist;
                                MarkPath(adj);
                                TrimSearchList();
                                }
                            }
                        }
                    }
            RemoveFromSearchList(cur);
            }
        return(goalFound);
        }


VERTEX_NODE* VertexList::FindVertex(int t, int w, int l)
   {
   VERTEX_NODE* cur;

   for(cur = GetFirstVertex();
      cur != (VERTEX_NODE*)NULL;
      cur = GetNextVertex(cur))
      {
      if ((cur->t == t)
      && (cur->w == w)
      && (cur->l == l))
         return(cur);           // found it

      if ((cur->t >= t)
      && (cur->w >= w)
      && (cur->l >= l))
         break;                 // passed it - it's not in the list
      }
   return((VERTEX_NODE*)NULL);
   }


VERTEX_NODE* VertexList::GetFirstVertex()
   {
   return(vertexHead);
   }


VERTEX_NODE* VertexList::GetGoalVertex()
   {
   VERTEX_NODE*tmp;

   for(tmp = GetFirstVertex();
      tmp != (VERTEX_NODE*)NULL;
      tmp = GetNextVertex(tmp))
      {
            if (tmp->nodeType == GOAL)
         break;
      }
   return(tmp);
   }


VERTEX_NODE* VertexList::GetNextVertex(VERTEX_NODE*cur)
   {
   return(cur->next);
   }


VERTEX_NODE* VertexList::GetStartVertex()
   {
   VERTEX_NODE*tmp;

   for(tmp = GetFirstVertex();
      tmp != (VERTEX_NODE*)NULL;
      tmp = GetNextVertex(tmp))
      {
```

```
			if (tmp->nodeType == START)
		break;
	    }
	return(tmp);
	}


 void VertexList::InsertNewVertex(VERTEX_NODE*a)
	{
	VERTEX_NODE *cur;

	if (a == (VERTEX_NODE*)NULL)
		return;

	if (vertexHead == (VERTEX_NODE*)NULL)
		{
		vertexHead = a;
		(*a).prev = (*a).next = (VERTEX_NODE*)NULL;
		return;
		}

	for(cur=vertexHead; cur != (VERTEX_NODE*)NULL;   cur=(*cur).next)
		{
		if ( (a->t > cur->t)
	|| ((a->t == cur->t)
	   && (a->w > cur->w))
	 || ((a->t == cur->t)
	   && (a->w == cur->w)
		&& (a->l > cur->l)) )
			{					// insert after cur
			if ((*cur).next == (VERTEX_NODE*)NULL)
				{
				(*a).next = (*cur).next;	// eol - append new node
				(*a).prev = cur;
				(*cur).next = a;
				break;
				}
			else
				continue;			// get next node
			}
		if ( (a->t < cur->t)
	|| ((a->t == cur->t)
	   && (a->w < cur->w))
	|| ((a->t == cur->t)
		   && (a->w == cur->w)
			&& (a->l < cur->l)) )
			{					// insert before cur
			if (cur->prev == (VERTEX_NODE*)NULL)
				{
				a->prev = (VERTEX_NODE*)NULL;// at start of list
				a->next = cur;
				cur->prev = a;
				vertexHead = a;
				break;
				}
			else
				{
				a->prev = cur->prev;		// in middle/end of list
				a->next = cur;
				cur->prev = a;
				cur = a->prev;
				cur->next = a;
				break;
				}
			}
		if ((a->t == cur->t)
	&& (a->w == cur->w)
		&& (a->l == cur->l))
			{					// insert after cur at eol
			if ((*cur).next == (VERTEX_NODE*)NULL)
				{
				(*a).next = (VERTEX_NODE*)NULL;
				(*a).prev = cur;
				(*cur).next = a;
				break;
				}
			else
				continue;
			}
		}
//	cout << "Inserted vertex (";
// cout << (*a).t << "," << (*a).w << "," << (*a).l << ")\n";
	}
```

```
void VertexList::ListAllVertices()
        {
        VERTEX_NODE *cur=vertexHead;

    while (cur != (VERTEX_NODE*)NULL)
        {
                cout << "Vertex:" <<  (*cur).nodeType << ": at (";
                cout << (*cur).t << "," << (*cur).w << "," << (*cur).l << ")\n";
                cur->edgeList->ListAllEdges();
        cur = (*cur).next;
        }
    }


void VertexList::ListSearchList()
        {
        VERTEX_NODE*cur;

        if (searchHead == (VERTEX_NODE*)NULL)
                {
                cout << "Empty SearchList! (marker=" << searchMarker << ")\n";
                return;
                }

        cout << "List of nodes in Search list (marker=" << searchMarker << ")\n";
        cur = searchHead;
        do
                {
                cout << "SearchList node (" << cur->t << "," << cur->w << "," << cur->l << ") ";
        cout << "Dist = " << cur->searchDist << "\n";
                cur=cur->searchNext;
                }
        while (cur != searchHead);
        cout << "End of SearchList\n";
        }


void VertexList::MarkPath(VERTEX_NODE*v)
        {
        VERTEX_NODE*tmp;

        if (v == (VERTEX_NODE*)NULL)
                return;

//      cout << "Path from Goal to Start\n";
        v->pathTo = (VERTEX_NODE*)NULL;
        do
                {
                tmp = v->pathFrom;
//              cout << "(" << v->t << "," << v->w << "," << v->l << ")\n";
                if (tmp != (VERTEX_NODE*)NULL)
                        {
                        tmp->pathTo = v;
                        v = v->pathFrom;
                        }
                }
        while (tmp != (VERTEX_NODE*)NULL);
        }


void VertexList::RemoveFromSearchList(VERTEX_NODE*a)
        {
        VERTEX_NODE *cur;

        if ((a->searchPrev == (VERTEX_NODE*)NULL)
        || (a->searchNext == (VERTEX_NODE*)NULL))
       {
       if (searchHead == a)
          searchHead = (VERTEX_NODE*)NULL;
       return;
       }

        cur=a->searchPrev;
        if (cur == a)
                {
       searchHead = (VERTEX_NODE*)NULL;
//              cout << "Removed Search Node (";
//     cout << a->t << "," << a->w << "," << a->l << "). SearchList empty\n";
                }
        else
                {
                cur->searchNext = a->searchNext;
                cur = a->searchNext;
```

```
            cur->searchPrev = a->searchPrev;
            if (searchHead == a)
                  searchHead = a->searchNext;
//          cout << "Removed Search Node (";
//    cout << a->t << "," << a->w << "," << a->l << ").\n";
            }
      a->searchPrev = a->searchNext = (VERTEX_NODE*)NULL;
      }


void VertexList::TrimSearchList()
   {
      VERTEX_NODE *cur, *tmpPtr=(VERTEX_NODE*)0;
      int trimmed=TRUE;

//    cout << "Trimming search list to " << searchTrimDist << " or less\n";
      if ((searchHead == (VERTEX_NODE*)NULL)
   || (searchTrimDist < 0.0))
            return;

   cur = searchHead;
      while ((tmpPtr != cur) || (trimmed == TRUE))
            {
            if (trimmed==TRUE)
                  {
                  tmpPtr=cur;
                  trimmed=FALSE;
                  }
            if ((*cur).searchDist >= searchTrimDist)
                  {
//                cout << "Trimmed out (";
//          cout << cur->t << "," << cur->w << "," << cur->l << ") ";
//          cout << "Dist was " << cur->searchDist << "\n";
                  if (cur == cur->searchNext)
                        {
           RemoveFromSearchList(cur);
                        break;
                        }
                  else
                        {
                        tmpPtr = cur->searchNext;
                        RemoveFromSearchList(cur);
                        cur = tmpPtr;
                        trimmed = TRUE;
                        }
                  }
            else
                  cur = cur->searchNext;
            }
//    cout << "-------\n";
   }
```

B.7. Partial Update Breadth First Graph Theory

The Partial Update Breadth First Search Algorithm for Graph Theory which was developed and implemented as part of this project was coded in ANSI C++. The filenames for each of the separate source code files were supplied inside C++ format comments (i.e. //) at the beginning of each file listing. A detailed explanation of the design behind this program was presented in Chapter 4.

```
// main.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <sys\timeb.h>
#include <dir.h>
#include <ctype.h>
#include "edge.hpp"
#include "vertex.hpp"
#include "object.hpp"
#include "domain.hpp"
#include "bench.hpp"
```

```
#define MAPFILE_MASK "MAP*.DAT"

int main(int, char**);
void BuildUpdateListFromDomain(VertexList&, VertexList&, Domain&);
void BuildVertexListFromDomain(VertexList&, Domain&);
void FindAllEdges(VertexList&, Domain&);
void FindEdgesForUpdateNode(VERTEX_NODE*, VertexList&, VertexList&, Domain&);
int MemStatus(char*);
void MergeLists(VertexList&, VertexList&, Domain&);
int SetWorkingDir(void);
void RenameMapFile(char*);


int main(int argc, char** argv)
	{
	char* shortName;
	struct ffblk mapfile;

	for(;;)
		{
		shortName = strstr(argv[0], "\\");
		if (shortName == (char*)NULL)
			{
			shortName = argv[0];
			cout << "This is " << shortName << "\n";
			break;
			}
		else
			argv[0] = shortName+1;
		}

	MemStatus("Free memory before mainloop in main: ");

	if (!SetWorkingDir())
		{
		cout << "Program exiting gracefully\n";
		return(1);
		}
	if (findfirst(MAPFILE_MASK, &mapfile, 0))
		{
		cout << "Program exiting gracefully - no map files found\n";
		return(1);
		}
	do
		{
		char tmp[256];
		int i;
		Benchmark stopWatch(shortName);

		sprintf(tmp, "Starting on %s: ", mapfile.ff_name);
		MemStatus(tmp);
		for (i=0; i<NUMTIMES; i++)
			{
			int timeTaken;
			float distTravelled;
			Domain theWorld(10, 20, 20, mapfile.ff_name);
			VertexList vertList, updateList;

			timeTaken = 0;
			distTravelled = 0.0;
			stopWatch.IterStart(i, mapfile.ff_name);
			stopWatch.Click();
			BuildVertexListFromDomain(vertList, theWorld);
			FindAllEdges(vertList, theWorld);
			for(;;)
				{
				int pathFoundFlag, goalFoundFlag, a, b, c;
				float dist;
				VERTEX_NODE *t1;

				stopWatch.Click();
				theWorld.DrawDomain();
				stopWatch.Click();
				pathFoundFlag = vertList.FindPath();
				stopWatch.Click();

				if (pathFoundFlag == TRUE)
					{
					cout << "Path found to the GOAL!";
					cout << "(dist=" << distTravelled << ")\n";
					}
				else
					{
					cout << "No path found to the GOAL!";
```

```
					cout << "(dist=" << distTravelled << ")\n";
					}

				t1 = vertList.GetStartVertex();
				if (pathFoundFlag)
					{
					a = t1->pathTo->t;
					b = t1->pathTo->w;
					c = t1->pathTo->l;
					}
				else
					{
					a = (t1->t)+1;
					b = t1->w;
					c = t1->l;
					}
				goalFoundFlag = theWorld.MoveRobot(t1->t, t1->w, t1->l,
a, b, c, &dist);
				distTravelled += dist;
				timeTaken++;
				if (goalFoundFlag)
					{
					cout << "MADE IT TO THE GOAL!\n";
					break;
					}

				theWorld.AdvanceTime();
				stopWatch.Click();
				BuildUpdateListFromDomain(vertList, updateList, theWorld);
				MergeLists(vertList, updateList, theWorld);
				}
			stopWatch.IterStop(timeTaken, distTravelled);
			}
		stopWatch.LogCalcs();
		RenameMapFile(mapfile.ff_name);
		}
	while (!findnext(&mapfile));

	MemStatus("Free memory after mainloop in main: ");
	return(0);
	}


void BuildUpdateListFromDomain(VertexList &vList, VertexList &uList, Domain &domain)
	{
	char type;
	int t=domain.GetDomainTimeSlices(), a;
	int w=domain.GetDomainWidth(), b;
	int l=domain.GetDomainLength(), c;
	VERTEX_NODE*newPtr;

	for(a=0; a<t; a++)
	  {
		for(b=0; b<w; b++)
			{
			for(c=0; c<l; c++)
				{
				type = domain.GetPointType(a, b, c);
				if ((type == VERTEX)
				|| (type == START)
				|| (type == GOAL))
					{
					VERTEX_NODE *tmpv, *tmpu;

					tmpv = vList.FindVertex(a, b, c);
					tmpu = uList.FindVertex(a, b, c);

					if ((!tmpv)
					&& (!tmpu))
						{
						newPtr = uList.BuildNewVertex(a, b, c, type);
						uList.InsertNewVertex(newPtr);
						continue;
						}

					if ((tmpv != (VERTEX_NODE*)NULL)
					&& (tmpv->nodeType != type))
						{
						vList.RemoveVertex(tmpv);
						vList.DelVertex(tmpv);
						newPtr = uList.BuildNewVertex(a, b, c, type);
						uList.InsertNewVertex(newPtr);
```

```
                                        continue;
                                        }

                                if ((tmpu != (VERTEX_NODE*)NULL)
                                && (tmpu->nodeType != type))
                                        {
                                        uList.RemoveVertex(tmpu);
                                        uList.DelVertex(tmpu);
                                        newPtr = uList.BuildNewVertex(a, b, c, type);
                                        uList.InsertNewVertex(newPtr);
                                        continue;
                                        }
                                }
                        else
                                {
                                VERTEX_NODE *v;

                                v = vList.FindVertex(a, b, c);
                                if (v!=(VERTEX_NODE*)NULL)
                                        {
                                        EDGE_NODE*e;

                                        for(e = v->edgeList->GetFirstEdge();
                                                e != (EDGE_NODE*)NULL;
                                                e = v->edgeList->GetNextEdge(e))
                                                {
                                                VERTEX_NODE* v2;

                                                v2 = vList.FindVertex(e->t, e->w, e->l);
                                                if (v2 != (VERTEX_NODE*)NULL)
                                                        {
                                                        vList.RemoveVertex(v2);
                                                        uList.InsertNewVertex(v2);
                                                        v2->edgeList->DelAllEdges();
                                                        }
                                                }
                                        vList.DelVertex(v);
                                        }
                                v = uList.FindVertex(a, b, c);
                                if (v != (VERTEX_NODE*)NULL)
                                        {
                                        uList.DelVertex(v);
                                        }
                                }
                        }
                }
        }
}

void BuildVertexListFromDomain(VertexList &vList, Domain &domain)
   {
        char type;
        int t=domain.GetDomainTimeSlices(), a;
        int w=domain.GetDomainWidth(), b;
        int l=domain.GetDomainLength(), c;
        VERTEX_NODE*newPtr;

        for(a=0; a<t; a++)
          {
                for(b=0; b<w; b++)
                        {
                        for(c=0; c<l; c++)
                                {
                                type = domain.GetPointType(a, b, c);
                                if ((type == VERTEX)
                                || (type == START)
                                || (type == GOAL))
                                        {
                                newPtr = vList.BuildNewVertex(a, b, c, type);
                                vList.InsertNewVertex(newPtr);
                                        }
                                }
                        }
                }
        }

void FindAllEdges(VertexList &vList, Domain &domain)
        {
        EDGE_NODE*e;
        VERTEX_NODE*a,*b;
        float dist;
```

```
    for(a = vList.GetFirstVertex();
            a != (VERTEX_NODE*)NULL;
            a = vList.GetNextVertex(a))
            {
            for(b = vList.GetNextVertex(a);
                    b != (VERTEX_NODE*)NULL;
                    b = vList.GetNextVertex(b))
      {
                    if (a == b)
                            continue;

                    if ((a->t == b->t)
                    && (a->w == b->w)
                    && (a->l == b->l))
                            continue;

      dist = domain.CheckLine(a->t, a->w, a->l, b->t, b->w, b->l);
      if (dist > 0.0)
         {
                            e = a->edgeList->BuildNewEdge(b->t, b->w, b->l, dist);
                            a->edgeList->InsertNewEdge(e);
                            }
                    dist = domain.CheckLine(b->t, b->w, b->l, a->t, a->w, a->l);
                    if (dist > 0.0)
                            {
                            e = b->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                            b->edgeList->InsertNewEdge(e);
                            }
                    }
            }
    }


void FindEdgesForUpdateNode(VERTEX_NODE*cur, VertexList &vList, VertexList &uList, Domain &domain)
        {
        EDGE_NODE*e;
        VERTEX_NODE*a;
        float dist;

        for(a = vList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = vList.GetNextVertex(a))
                {
        if (a == cur)
                continue;

                if ((a->t == cur->t)
                && (a->w == cur->w)
                && (a->l == cur->l))
                        continue;

        dist = domain.CheckLine(a->t, a->w, a->l, cur->t, cur->w, cur->l);
        if (dist > 0.0)
           {
                        e = a->edgeList->BuildNewEdge(cur->t, cur->w, cur->l, dist);
                        a->edgeList->InsertNewEdge(e);
                        }
                dist = domain.CheckLine(cur->t, cur->w, cur->l, a->t, a->w, a->l);
                if (dist > 0.0)
                        {
                        e = cur->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                        cur->edgeList->InsertNewEdge(e);
                        }
                }

        for(a = uList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = uList.GetNextVertex(a))
                {
        if (a == cur)
                continue;

                if ((a->t == cur->t)
                && (a->w == cur->w)
                && (a->l == cur->l))
                        continue;

        dist = domain.CheckLine(a->t, a->w, a->l, cur->t, cur->w, cur->l);
        if (dist > 0.0)
                        {
                        e = a->edgeList->BuildNewEdge(cur->t, cur->w, cur->l, dist);
                        a->edgeList->InsertNewEdge(e);
                        }
                dist = domain.CheckLine(cur->t, cur->w, cur->l, a->t, a->w, a->l);
```

```
            if (dist > 0.0)
                  {
                  e = cur->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                  cur->edgeList->InsertNewEdge(e);
        }
            }
      }


int MemStatus(char *StatusMessage)
      {
      char tmp[256];
      fstream debugFile;
      long MemLeft;
      int ret;

      debugFile.open("DEBUG.LOG", ios::app);
      if (!debugFile)
            cout << "Unable to open DEBUG.LOG\n";

      MemLeft = (long) coreleft();
      sprintf(tmp, "%s%ld\n", StatusMessage, MemLeft);
      debugFile.write(tmp, strlen(tmp));
      debugFile.close();
//    cout << StatusMessage << MemLeft << "\n";

      ret = farheapcheck();
      if (ret == _HEAPOK)
            cout << "Heap ok" << StatusMessage << "\n";
      else
            {
            cout << "Heap error <" << ret << ">" << StatusMessage<< "\n";
            getch();
            return(FALSE);
            }
      return(TRUE);
      }


void MergeLists(VertexList &vList, VertexList &uList, Domain &domain)
   {
   VERTEX_NODE* cur;

   for (cur=uList.GetFirstVertex();
      cur!=(VERTEX_NODE*)NULL;
      cur=uList.GetFirstVertex())
      {
      uList.RemoveVertex(cur);
            FindEdgesForUpdateNode(cur, vList, uList, domain);
            vList.InsertNewVertex(cur);
            }
       cur = vList.GetStartVertex();
       if (cur != (VERTEX_NODE*)NULL)
            {
            cur->searchMarker = vList.GetSearchMarker() + 1;
            vList.AddToSearchList(cur);
            }
      }


int SetWorkingDir()
      {
      char mapdir[256];

      cout <<"Enter the directory containing map files, or \"q\" for quit:";
      cin >> mapdir;
      if (toupper(mapdir[0]) == 'Q')
            {
            cout << "Quitting...\n";
            return(FALSE);
            }
      if (chdir(mapdir))
            {
            cout << "The directory " << mapdir << " could not be found.\n";
            return(FALSE);
            }
      cout << "Made " << mapdir << " the current directory.\n";
      return(TRUE);
      }

void RenameMapFile(char*filename)
```

```
        {
        char newfilename[128];
        char* ch;
        int i=(int)'.';

        strcpy(newfilename, filename);
        ch = strrchr(newfilename,i);
        if (ch != (char*)NULL)
                {
                strcpy(ch, ".bak");
                rename(filename, newfilename);
                }
        else
                exit(1);
        }

// bench.hpp
#define FALSE 0
#define TRUE !FALSE
#define NUMTIMES 10
#define MAXFILENAME 13

class Benchmark
        {
        public:
                Benchmark(char*);
                ~Benchmark();
                void Click(void);
                void IterStart(int, char*);
                void IterStop(int, float);
                void LogCalcs(void);
        private:
                void Diff(struct timeb*, struct timeb*, struct timeb*);

                fstream logFile;
                fstream avgFile;
                char mapFileName[MAXFILENAME];
                struct timeb benchmarks[NUMTIMES][2];
                float distRobotTravelled[NUMTIMES];
                int timeTaken[NUMTIMES];
                int currentIter;
                int clickToggleFlag;
                struct timeb elapsedTime, computeTime;
                struct timeb startTime, clickOnTime, clickOffTime;
        };

// bench.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <sys\timeb.h>
#include "bench.hpp"


Benchmark::Benchmark(char *fileName)
        {
        char *tmp;
        char logname[MAXFILENAME];

        while((tmp = strchr(fileName, '\\')) != (char*)NULL)
                fileName = tmp+1;
        strcpy(logname, fileName);
        if ((tmp = strchr(logname, '.')) != (char*)NULL)
                strcpy(tmp, ".LOG");
        else
                strcat(logname, ".LOG");
        logFile.open(logname, ios::app);
        if (!logFile)
                cout << "Unable to open " << logname << "\n";

        if ((tmp = strchr(logname, '.')) != (char*)NULL)
                strcpy(tmp, ".AVG");
        else
                strcat(logname, ".AVG");
        avgFile.open(logname, ios::app);
        if (!avgFile)
                cout << "Unable to open " << logname << "\n";

        cout << "Initialised Benchmark class\n";
        }
```

```
Benchmark::~Benchmark()
	{
	if (logFile)
		{
		logFile.flush();
		logFile.close();
		}
	if (avgFile)
		{
		avgFile.flush();
		avgFile.close();
		}
	cout << "Closed log files and Destroying Benchmark class\n";
	}


void Benchmark::Click()
	{
	switch(clickToggleFlag)
		{
		case TRUE:
			ftime(&clickOnTime);
			clickToggleFlag = FALSE;
			break;
		case FALSE:
		default:
			ftime(&clickOffTime);
			Diff(&clickOnTime, &clickOffTime, &computeTime);
			clickToggleFlag = TRUE;
		}
	}


void Benchmark::Diff(struct timeb*start, struct timeb*stop, struct timeb*diff)
	{
	if ((*stop).millitm < (*start).millitm)
		{
		(*stop).millitm += (short)1000; /* carry when subtracting, stops*/
		(*start).time += 1L;            /* negative wraparound problems!*/
		}
	(*diff).millitm += (*stop).millitm - (*start).millitm;
	(*diff).time += (long)((*diff).millitm / (short)1000);
	(*diff).millitm %= (short)1000;
	(*diff).time += ((*stop).time-(*start).time);
	}


void Benchmark::IterStart(int i, char*s)
	{
	currentIter = i;
	clickToggleFlag = TRUE;
	computeTime.time = elapsedTime.time = 0L;
	computeTime.millitm = elapsedTime.millitm = 0;

	strcpy(mapFileName, s);

	ftime(&startTime);
	}


void Benchmark::IterStop(int t, float distTravelled)
	{
	struct timeb stopTime;

	ftime(&stopTime);
	Diff(&startTime, &stopTime, &elapsedTime);

	benchmarks[currentIter][0].time = elapsedTime.time;
	benchmarks[currentIter][0].millitm = elapsedTime.millitm;
	benchmarks[currentIter][1].time = computeTime.time;
	benchmarks[currentIter][1].millitm = computeTime.millitm;
	timeTaken[currentIter] = t;
	distRobotTravelled[currentIter] = distTravelled;
	}



void Benchmark::LogCalcs()
	{
	char tmp[256];
	float avgDist;
	struct timeb avg;
```

```
	int i, avgTimeTaken;

	for(i=0, avg.time=0L, avg.millitm=0, avgDist=0.0, avgTimeTaken=0;
		i<NUMTIMES;
		i++)
		{
		sprintf(tmp, "%s (%02d) Elapsed time:%05ld.%03d\n", mapFileName, i,
				benchmarks[i][0].time, benchmarks[i][0].millitm);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Compute time:%05ld.%03d\n", mapFileName, i,
				benchmarks[i][1].time, benchmarks[i][1].millitm);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Dist travelled:%f\n", mapFileName, i,
				distRobotTravelled[i]);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Time Slices Taken:%d\n", mapFileName, i,
				timeTaken[i]);
		logFile.write(tmp, strlen(tmp));
		avg.time+=benchmarks[i][0].time;
		avg.millitm+=benchmarks[i][0].millitm;
		if (avg.millitm % 1000 != avg.millitm)
			{
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;
			}
		}

	sprintf(tmp, "%s Tot. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));
	cout << avg.time << "." << avg.millitm << "\n";
	cout << tmp << "\n";

	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;

	sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	avgFile.write(tmp, strlen(tmp));
//	sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));

	for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
		{
		avg.time+=benchmarks[i][1].time;
		avg.millitm+=benchmarks[i][1].millitm;
		if (avg.millitm % 1000 != avg.millitm)
			{
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;
			}
		}
	sprintf(tmp, "%s Tot. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));
	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;
	sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	avgFile.write(tmp, strlen(tmp));
//	sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));

	for(i=0, avgTimeTaken=0; i<NUMTIMES; i++)
		avgTimeTaken+=timeTaken[i];
	avgTimeTaken = avgTimeTaken / NUMTIMES;
	sprintf(tmp, "%s Avg. Time Taken:%d\n", mapFileName, avgTimeTaken);
	avgFile.write(tmp, strlen(tmp));
	logFile.write(tmp, strlen(tmp));

	for(i=0, avgDist=0.0; i<NUMTIMES; i++)
		avgDist += distRobotTravelled[i];
	avgDist = avgDist / ((float)NUMTIMES);
	sprintf(tmp, "%s Avg. Dist Travelled:%f\n", mapFileName, avgDist);
	avgFile.write(tmp, strlen(tmp));
	logFile.write(tmp, strlen(tmp));

	sprintf(tmp, "%s ====================\n", mapFileName);
	avgFile.write(tmp, strlen(tmp));
//	sprintf(tmp, "%s ====================\n", mapFileName);
	logFile.write(tmp, strlen(tmp));
	}

// domain.hpp
```

```
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00

#define FROM_FRONT 0
#define FROM_BACK 1
#define FROM_LEFT 2
#define FROM_RIGHT 3
#define FROM_FRONTRIGHT 4
#define FROM_FRONTLEFT 5
#define FROM_BACKRIGHT 6
#define FROM_BACKLEFT 7
#define FROM_UP 8
#define FROM_DOWN 9
#define FROM_NOWHERE 10

#define NOCOST (float)0.0
#define NORMAL (float)1.0
#define NORMAL_DIAG (float)1.414214
#define BLOCKED (float)1000.0

typedef struct
	{
	float dist;
	int from;
	char type;
	float cost[9];
	} POINT;

class Domain
	{
	public:
		Domain(int, int, int, char*);
		~Domain();
		void AdvanceTime(void);
	float CheckLine(int, int, int, int, int, int);
		void DrawDomain(void);
	int GetDomainLength(void) { return(domainLength); }
	int GetDomainTimeSlices(void) { return(domainTimeSlices); }
	int GetDomainWidth(void) { return(domainWidth); }
		POINT* GetPoint(int, int, int);
	float GetPointCost(int, int, int, int);
		char GetPointFrom(int, int, int);
		char GetPointType(int, int, int);
		int IsPointClear(int, int, int);
		int IsPointGoal(int, int, int);
		int IsPointNearObject(int, int, int);
		int IsPointObject(int, int, int);
		int IsPointStart(int, int, int);
	int IsPointVertex(int, int, int);
		int MoveRobot(int, int, int, int, int, int, float*);
		void SetPointFrom(int, int, int, int);
		void SetPointType(int, int, int, char);
	private:
		void AgeTimeSlices(void);
	int CalcRobotDir(int, int, int, int, int, int);
		int ClearAdjPointOK(int, int, int);
		void ClearMobileObject(int, int, int);
		void ClearVerticesInTimeSlice(int);
		void DrawTimeSlice(int);
		void InitTimeSlice(int);
		void MarkMobileObject(int, int, int);
		void MoveMobileObject(OBJECT_NODE*);
		void MoveMobileObjects(void);
		void SetAdjObjsInTimeSlice(int);
		void SetGoalFromFile(char*);
		void SetMobileObjsFromFile(char*);
		void SetPermObjsInTimeSlice(int);
		void SetStartFromFile(char*);
		void SetVerticesInTimeSlice(int);
```

```
                ObjectList objList;
                POINT** domainHead;
        int domainWidth;
        int domainLength;
        int domainTimeSlices;
        };

// domain.cpp
#include <iostream.h>
#include <fstream.h>
#include <alloc.h>          // for coreleft()
#include <stdio.h>
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <conio.h>
#include "object.hpp"
#include "domain.hpp"

Domain::Domain(int numTimeSlices, int width, int length, char*mapFileName)
        {
        int i;

        cout << "Contructing Domain class\n";
        domainTimeSlices = numTimeSlices;
        domainWidth = width;
        domainLength = length;

        cout << "FreeHeap:" << farcoreleft() << "\n";
        cout << "Amount needed for one timeslice:";
        cout << width*length*sizeof(POINT) << "\n";

        domainHead = (POINT**)farcalloc(numTimeSlices, sizeof(POINT**));
        if (domainHead == (POINT**)NULL)
                {
                cout << "Not enough memory to build model of world\n";
                domainHead = (POINT***)NULL;
                domainTimeSlices = domainWidth = domainLength = 0;
                return;
                }

        for(i=0; i<numTimeSlices; i++)
                {
                domainHead[i] = (POINT*)farcalloc(width*length, sizeof(POINT));
                cout << "FreeHeap after timeslice allocated:" << farcoreleft() << "\n";
                if (domainHead[i] == (POINT*)NULL)
                        {
                        if (i==0)
                                cout << "Not enough memory to build model of world\n";
                        else
                                cout << "Only enough memory to build " << i << " of the ";
                                cout << numTimeSlices << " timeslices in model of world\n";
                        domainTimeSlices = i;
                        break;
                        }
                InitTimeSlice(i);
                SetPermObjsInTimeSlice(i);
                }
        SetStartFromFile(mapFileName);
        SetGoalFromFile(mapFileName);
        SetMobileObjsFromFile(mapFileName);
        for(i=0; i<domainTimeSlices; i++)
                {
                SetAdjObjsInTimeSlice(i);
                SetVerticesInTimeSlice(i);
                }
        for(i=0; i<domainTimeSlices-1; i++)
                AdvanceTime();
        cout << "FreeHeap after Domain allocated:" << farcoreleft() << "\n";
        }


Domain::~Domain()
        {
        POINT *tmp;
        int i;

        cout << "Destructing Domain class\n";
        if (domainHead == (POINT**)NULL)
                {
                cout << "Not freeing domain - DomainHead was NULL\n";
                return;
                }
        for(i=0; i<domainTimeSlices; i++)
```

```
		{
		tmp = domainHead[i];
		farfree(tmp);
		}
	farfree(domainHead);
	domainHead = (POINT**)NULL;
	cout << "FreeHeap after Domain deallocated:" << farcoreleft() << "\n";
//	getch();
	}


void Domain::AdvanceTime()
	{
	AgeTimeSlices();
//	DrawDomain();
	ClearVerticesInTimeSlice(domainTimeSlices-1);
	MoveMobileObjects();
//	DrawDomain();
	SetVerticesInTimeSlice(domainTimeSlices-1);
//	DrawDomain();
	}


//////////////////////////////////////
//
// Copy the contents of every timeslice
// into the previous timeslice
// [0] <= [1], [1] <= [2], etc.
// Leave the last timeslice unchanged.
// Another routine will decide the moves
// for all the mobile objects.
//
// A quick way to do this is moving ptrs
// to the timeslices and only copying the
// contents of the last timeslice over the
// contents of the first timeslice
//
//////////////////////////////////////
void Domain::AgeTimeSlices()
	{
	int i, j;
	POINT *tmp;

	tmp = domainHead[0];
	for(i=0, j=1; i<domainTimeSlices-1; i++, j++)
		domainHead[i] = domainHead[j];
	domainHead[domainTimeSlices-1] = tmp;
	_fmemcpy(domainHead[domainTimeSlices-1],
				domainHead[domainTimeSlices-2],
				sizeof(*domainHead[domainTimeSlices-1]));
	}


int Domain::CalcRobotDir(int st, int sw, int sl, int et, int ew, int el)
	{
	if ((!IsPointStart(st, sw, sl))
	|| (IsPointGoal(st, sw, sl)))
		return(clear);

	if ((ew - sw > 0)
	&& (el - sl > 0))
		{
		sw++, sl++;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(frontright);
		else
			return(clear);
		}

	if ((ew - sw > 0)
	&& (el - sl == 0))
		{
		sw++;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(right);
		else
			return(clear);
		}

	if ((ew - sw > 0)
```

```
&& (el - sl < 0))
        {
        sw++, sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(backright);
        else
                return(clear);
        }

if ((ew - sw == 0)
&& (el - sl > 0))
        {
        sl++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(front);
        else
                return(clear);
        }

if ((ew - sw == 0)
&& (el - sl == 0)
&& (et - st > 0))
        {
        st++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(up);
        else
                return(clear);
        }

if ((ew - sw == 0)
&& (el - sl < 0))
        {
        sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(back);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl > 0))
        {
        sw--, sl++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(frontleft);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl == 0))
        {
        sw--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(left);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl < 0))
        {
        sw--, sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(backleft);
        else
                return(clear);
        }
return(clear);
```

```
	}


float Domain::CheckLine(int st, int sw, int sl, int et, int ew, int el)
	{
//	static int displayCounter=1;
	int tmpt, tmpw, tmpl;
	float dist;

	for(dist=0.0, tmpt=et-st, tmpw=ew-sw, tmpl=el-sl;
		(tmpt != 0) || (tmpw != 0) || (tmpl != 0);
		tmpt=et-st, tmpw=ew-sw, tmpl=el-sl)
		{
		if (tmpt < 0)
			{
			return(-1.0);
			}
		if ((tmpw > 0)
		&& (tmpl > 0))
			{
			dist += GetPointCost(st, sw++, sl++, frontright);
			if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
				return(-1.0);
	  continue;
			}
	 if ((tmpw > 0)
		&& (tmpl == 0))
			{
	  dist += GetPointCost(st, sw++, sl, right);
		if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
			}
		if ((tmpw > 0)
		&& (tmpl < 0))
			{
	  dist += GetPointCost(st, sw++, sl--, backright);
			if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
		if ((tmpw < 0)
		&& (tmpl > 0))
			{
	  dist += GetPointCost(st, sw--, sl++, frontleft);
		if ((!IsPointClear(st, sw, sl))
		&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
		if ((tmpw < 0)
		&& (tmpl == 0))
			{
	  dist += GetPointCost(st, sw--, sl, left);
		if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
		if ((tmpw < 0)
		&& (tmpl < 0))
			{
	  dist += GetPointCost(st, sw--, sl--, backleft);
			if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
				return(-1.0);
	  continue;
			}
		if ((tmpw == 0)
		&& (tmpl > 0))
			{
	  dist += GetPointCost(st, sw, sl++, front);
		if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
			return(-1.0);
	  continue;
	  }
		if ((tmpw == 0)
		&& (tmpl < 0))
			{
```

```
		dist += GetPointCost(st, sw, sl--, back);
			if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
				return(-1.0);
		continue;
		}
	if ((tmpw == 0)
			&& (tmpl == 0)
			&& (tmpt >0))
				{
		dist += GetPointCost(st++, sw, sl, up);
			if ((!IsPointClear(st, sw, sl))
			&& ((st != et) || (sw != ew) || (sl != el)))
				return(-1.0);
		continue;
		}
			}
//	cout << "CheckLine:" << displayCounter++ << ":returned" << dist << "\n";
	return(dist);
	}


int Domain::ClearAdjPointOK(int t, int w, int l)
	{
	char type;
	int a,b;

	for(a=w-1;a<=w+1;a++)
		{
		if ((a < 0) || (a >= domainWidth))
			continue;
		for(b=l-1;b<=l+1;b++)
			{
			if ((b < 0) || (b >= domainLength))
				continue;
			if ((a == w) && (b == l))
				continue;

	  type = GetPointType(t, a, b);
			if ((type == OBJECT)
			||  (type == MOBILE_OBJECT))
				return(FALSE);
			}
		}
	return(TRUE);
	}


void Domain::ClearMobileObject(int t, int w, int l)
   {
	int i, j;

	if (GetPointType(t, w, l) == MOBILE_OBJECT)
		{
		if (ClearAdjPointOK(t, w, l))
			{
			SetPointType(t, w, l, CLEAR);
			SetPointFrom(t, w, l, FROM_NOWHERE);
			}
		else
			{
			SetPointType(t, w, l, ADJ_TO_OBJECT);
			SetPointFrom(t, w, l, FROM_NOWHERE);
			}
		}
	for(i=w-1, j=l-1; j<l+2;)
		{
      if ((GetPointType(t, i, j) == ADJ_TO_OBJECT)
		&& (ClearAdjPointOK(t, i, j)))
		{
		SetPointType(t, i, j, CLEAR);
			SetPointFrom(t, i, j, FROM_NOWHERE);
			}
		if (i == w+1)
			{
			i = w-1;
			j++;
			}
		else
			i++;
		}
	}
```

```
void Domain::ClearVerticesInTimeSlice(int t)
	{
	int w, l, pointType;

	for(w=0; w<domainWidth; w++)
		{
		for(l=0; l<domainLength; l++)
			{
			pointType = GetPointType(t, w, l);
			if (pointType == VERTEX)
				{
				SetPointFrom(t, w, l, FROM_NOWHERE);
				SetPointType(t, w, l, CLEAR);
				continue;
				}
			if (pointType == MOBILE_OBJECT)
				{
	ClearMobileObject(t, w, l);
				}
			}
		}
	}


void Domain::DrawDomain()
	{
	int i;

	for (i=0; i< domainTimeSlices; i++)
		{
		DrawTimeSlice(i);
//		getch();
		}
	}


void Domain::DrawTimeSlice(int timeSlice)
	{
	char type;
	int a,b,y;

	clrscr();
	gotoxy(1,1);
	cout << "time=t+" << timeSlice;

	for(b=0, y=2;b<domainLength; b++, y++)
		{
		gotoxy(1, y);
		for(a=0; a<domainWidth; a++)
			{
			type = GetPointType(timeSlice, a, b);
			switch(type)
				{
				case GOAL:
				case START:
				case ADJ_TO_OBJECT:
				case OBJECT:
				case MOBILE_OBJECT:
				case VERTEX:
					cout << type;
					break;
				case CLEAR:
					switch(GetPointFrom(timeSlice, a, b))
						{
						case FROM_RIGHT:
							cout << "R";
							break;
						case FROM_LEFT:
							cout << "L";
							break;
						case FROM_UP:
							cout << "U";
							break;
						case FROM_DOWN:
							cout << "D";
							break;
						case FROM_FRONT:
							cout << "F";
							break;
						case FROM_BACK:
							cout << "B";
							break;
						case FROM_BACKLEFT:
							cout << "T";
```

```
                                        break;
                                case FROM_BACKRIGHT:
                                        cout << "U";
                                        break;
                                case FROM_FRONTLEFT:
                                        cout << "V";
                                        break;
                                case FROM_FRONTRIGHT:
                                        cout << "W";
                                        break;
                                case FROM_NOWHERE:
                                default:
                                        cout << ".";
                                        break;
                                }
                        break;
                default:
                        cout << "?";
                        break;
                }
            }
        cout << "\n";
        }
    }


POINT* Domain::GetPoint(int timeSlice, int width, int length)
    {
   POINT *tmp, *tmp2;

   tmp = domainHead[timeSlice];
       tmp2 = tmp + (domainWidth*width) + length;
   return(tmp2);
   }


float Domain::GetPointCost(int timeSlice, int width, int length, int dir)
   {
   POINT *tmp;

   tmp = GetPoint(timeSlice, width, length);
       return(tmp->cost[dir]);
       }

char Domain::GetPointFrom(int timeSlice, int width, int length)
   {
   POINT *tmp;

   tmp = GetPoint(timeSlice, width, length);
       return(tmp->from);
   }


char Domain::GetPointType(int timeSlice, int width, int length)
       {
   POINT *tmp;

       tmp = GetPoint(timeSlice, width, length);
       return(tmp->type);
       }


void Domain::InitTimeSlice(int timeSlice)
       {
       POINT*tmp;
   int a,b;

       for(a=0;a<domainWidth;a++)
               {
               for(b=0;b<domainLength;b++)
                       {
          tmp = GetPoint(timeSlice, a, b);
                       tmp->dist=0.0;
                       tmp->from=FROM_NOWHERE;
                       tmp->type=CLEAR;

/*                     if (timeSlice == domainTimeSlices-1)
                               tmp->cost[FROM_UP] = BLOCKED;
                       else*/
                               tmp->cost[FROM_UP] = NORMAL;

                       if ((a == 0) && (b == 0))
                               {
                               tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
```

```
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
        tmp->cost[FROM_BACK] = BLOCKED;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = BLOCKED;
        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a == 0) && (b < domainLength-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = BLOCKED;
        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a == 0) && (b==domainLength-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = BLOCKED;
        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
        tmp->cost[FROM_FRONT] = BLOCKED;
        continue;
        }
if ((a < domainWidth-1) && (b == 0))
        {
        tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
        tmp->cost[FROM_BACK] = BLOCKED;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a < domainLength-1) && (b < domainWidth-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a < domainWidth-1) && (b == domainLength-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
        tmp->cost[FROM_FRONT] = BLOCKED;
        continue;
        }
if ((a == domainWidth-1) && (b == 0))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
        tmp->cost[FROM_RIGHT] = BLOCKED;
        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
        tmp->cost[FROM_BACK] = BLOCKED;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a == domainWidth-1) && (b < domainWidth-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
```

```
                    tmp->cost[FROM_RIGHT] = BLOCKED;
                    tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                    tmp->cost[FROM_BACK] = NORMAL;
                    tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                    tmp->cost[FROM_LEFT] = NORMAL;
                    tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                    tmp->cost[FROM_FRONT] = NORMAL;
                    continue;
                    }
                if ((a == domainLength-1) && (b == domainWidth-1))
                    {
                    tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                    tmp->cost[FROM_RIGHT] = BLOCKED;
                    tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                    tmp->cost[FROM_BACK] = NORMAL;
                    tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                    tmp->cost[FROM_LEFT] = NORMAL;
                    tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                    tmp->cost[FROM_FRONT] = BLOCKED;
                    continue;
                    }
                }
            }
        }


int Domain::IsPointClear(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
            return(FALSE);
        switch(GetPointType(t, w, l))
            {
            case CLEAR:
                return(TRUE);
            default:
                return(FALSE);
            }
        }


int Domain::IsPointGoal(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
            return(FALSE);
        switch(GetPointType(t, w, l))
            {
            case GOAL:
                return(TRUE);
            default:
                return(FALSE);
            }
        }



int Domain::IsPointNearObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
            return(TRUE);
        switch(GetPointType(t, w, l))
            {
            case OBJECT:
                return(OBJECT);
            case MOBILE_OBJECT:
                return(MOBILE_OBJECT);
            case ADJ_TO_OBJECT:
```

```
                        return(ADJ_TO_OBJECT);
                default:
                        return(FALSE);
                }
        }

int Domain::IsPointObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
                case OBJECT:
                        return(OBJECT);
                case MOBILE_OBJECT:
                        return(MOBILE_OBJECT);
                case ADJ_TO_OBJECT:
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointStart(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case START:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointVertex(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case VERTEX:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


void Domain::MarkMobileObject(int t, int w, int l)
        {
        int i, j;

        if ((GetPointType(t, w, l) == CLEAR)
        || (GetPointType(t, w, l) == ADJ_TO_OBJECT))
                {
                SetPointType(t, w, l, MOBILE_OBJECT);
                SetPointFrom(t, w, l, FROM_NOWHERE);
                }
        for(i=w-1, j=l-1; j<l+2;)
                {
                if (GetPointType(t, i, j) == CLEAR)
                        {
                        SetPointType(t, i, j, ADJ_TO_OBJECT);
                        SetPointFrom(t, i, j, FROM_NOWHERE);
                        }
```

```
			if (i == w+1)
				{
				i = w-1;
				j++;
				}
			else
				i++;
			}
		}

void Domain::MoveMobileObject(OBJECT_NODE*object)
	{
	int t=domainTimeSlices-1, w=(*object).w, l=(*object).l;

	ClearMobileObject(t, w, l);
	switch((*object).direction)
		{
		case front:
			if ((IsPointObject(t, w, l+1))
			|| (IsPointStart(t, w, l+1))
			|| (IsPointGoal(t, w, l+1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).l++;
			break;
		case frontleft:
			if ((IsPointObject(t, w-1, l+1))
			|| (IsPointStart(t, w-1, l+1))
			|| (IsPointGoal(t, w-1, l+1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).l++, (*object).w--;
			break;
		case left:
			if ((IsPointObject(t, w-1, l))
			|| (IsPointStart(t, w-1, l))
			|| (IsPointGoal(t, w-1, l)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w--;
			break;
		case backleft:
			if ((IsPointObject(t, w-1, l-1))
			|| (IsPointStart(t, w-1, l-1))
			|| (IsPointGoal(t, w-1, l-1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w--, (*object).l--;
			break;
		case back:
			if ((IsPointObject(t, w, l-1))
			|| (IsPointStart(t, w, l-1))
			|| (IsPointGoal(t, w, l-1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).l--;
			break;
		case backright:
			if ((IsPointObject(t, w+1, l-1))
			|| (IsPointStart(t, w+1, l-1))
			|| (IsPointGoal(t, w+1, l-1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w++, (*object).l--;
			break;
		case right:
			if ((IsPointObject(t, w+1, l))
			|| (IsPointStart(t, w+1, l))
			|| (IsPointGoal(t, w+1, l)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w++;
			break;
		case frontright:
			if ((IsPointObject(t, w+1, l+1))
			|| (IsPointStart(t, w+1, l+1))
			|| (IsPointGoal(t, w+1, l+1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w++, (*object).l++;
			break;
		default:
			cout << "*** unknown direction for mobile object ignored ***\n";
```

```
                        break;
                }
        MarkMobileObject(t, object->w, object->l);
        }


void Domain::MoveMobileObjects()
        {
        OBJECT_NODE*cur, *orig;

        orig = cur = objList.GetNextObject();
        if (cur == (OBJECT_NODE*)NULL)
        return;

        do
                {
                if ((*cur).velocity > 0)
                        MoveMobileObject(cur);

        cur = objList.GetNextObject();
                } while (cur != orig);
        }


int Domain::MoveRobot(int st, int sw, int sl, int et, int ew, int el, float*dist)
        {
        int dir;

        *dist = 0.0;
        if (!IsPointStart(st, sw, sl))
                {
                *dist = 0.0;
                return(FALSE);
                }

        dir = CalcRobotDir(st, sw, sl, et, ew, el);
        SetPointType(st, sw, sl, CLEAR);
        SetPointFrom(st, sw, sl, FROM_NOWHERE);

        switch(dir)
                {
                case frontright:
                        *dist = GetPointCost(st, sw, sl, frontright);
                        st++, sw++, sl++;
                        if ((!IsPointClear(st, sw, sl))
                        && (!IsPointVertex(st, sw, sl))
                        && (!IsPointGoal(st, sw, sl)))
                                {
                                st--, sw--, sl--;
                                *dist = 0.0;
                                }
                        break;
                case right:
                        *dist = GetPointCost(st, sw, sl, right);
                        st++, sw++;
                        if ((!IsPointClear(st, sw, sl))
                        && (!IsPointVertex(st, sw, sl))
                        && (!IsPointGoal(st, sw, sl)))
                                {
                                st--, sw--;
                                *dist = 0.0;
                                }
                        break;
                case backright:
                        *dist = GetPointCost(st, sw, sl, backright);
                        st++, sw++, sl--;
                        if ((!IsPointClear(st, sw, sl))
                        && (!IsPointVertex(st, sw, sl))
                        && (!IsPointGoal(st, sw, sl)))
                                {
                                st--, sw--, sl++;
                                *dist = 0.0;
                                }
                        break;
                case frontleft:
                        *dist = GetPointCost(st, sw, sl, frontleft);
                        st++, sw--, sl++;
                        if ((!IsPointClear(st, sw, sl))
                        && (!IsPointVertex(st, sw, sl))
                        && (!IsPointGoal(st, sw, sl)))
                                {
                                st--, sw++, sl--;
                                *dist = 0.0;
```

```
                    }
                break;
            case left:
                *dist = GetPointCost(st, sw, sl, left);
                st++, sw--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                    {
                    st--, sw++;
                    *dist = 0.0;
                    }
                break;
            case backleft:
                *dist = GetPointCost(st, sw, sl, backleft);
                st++, sw--, sl--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                    {
                    st--, sw++, sl++;
                    *dist = 0.0;
                    }
                break;
            case front:
                *dist = GetPointCost(st, sw, sl, front);
                st++, sl++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                    {
                    st--, sl--;
                    *dist = 0.0;
                    }
                break;
            case back:
                *dist = GetPointCost(st, sw, sl, back);
                st++, sl--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                    {
                    st--, sl++;
                    *dist = 0.0;
                    }
                break;
            case up:
                *dist = GetPointCost(st, sw, sl, up);
                st++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                    {
                    st--;
                    *dist = 0.0;
                    }
                break;
            default:        // no movement - start IS goal
                *dist = 0.0;
                break;
            }
        if (IsPointGoal(st, sw, sl))
            return(TRUE);
        else
            {
            SetPointType(st, sw, sl, START);
            SetPointFrom(st, sw, sl, FROM_NOWHERE);
            return(FALSE);
            }
        }

void Domain::SetAdjObjsInTimeSlice(int timeSlice)
        {
        int a,b;
        int w,l;

        for(w=0; w<domainWidth; w++)
            {
            for(l=0; l<domainLength; l++)
                {
                if (!IsPointObject(timeSlice, w, l))
                    continue;
```

```
                        for(a=w-1; a<=w+1; a++)
                                {
                                if ((a < 0) || (a >= domainWidth))
                                        continue;
                                for(b=l-1; b<=l+1; b++)
                                        {
                                        if ((b < 0) || (b >= domainLength))
                                                continue;

                                        if (GetPointType(timeSlice, a, b) == CLEAR)
                                                {
                                                SetPointFrom(timeSlice, a, b, FROM_NOWHERE);
                                                SetPointType(timeSlice, a, b, ADJ_TO_OBJECT);
                                                }

                                        }
                                }
                        }
                }

void Domain::SetGoalFromFile(char*fileName)
        {
        int i, w, l;
        char recType;
        char tmp[256];
        fstream dataFile;

        dataFile.open(fileName, ios::in);
        if (!dataFile)
                cout << "Unable to open " << fileName << ". No Goal added.\n";

        for(;;)
                {
                dataFile. getline(tmp, sizeof(tmp));
                if (strlen(tmp) == 0)
                        break;
                if (tmp[0] == GOAL)
                        {
//                      cout << "The GOAL line is: " << tmp << "\n";
                        if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                                {
                for(i=0; i<domainTimeSlices; i++)
                  {
                                SetPointFrom(i, w, l, FROM_NOWHERE);
                                SetPointType(i, w, l, GOAL);
                  }
                                break;
                                }
                        else
                                cout << "Improperly formatted line ignored\n";
                        }
                }
        dataFile.close();
        }


void Domain::SetMobileObjsFromFile(char*fileName)
        {
        int w, l;
        char recType;
        char tmp[256];
        fstream dataFile;
        OBJECT_NODE* newObj;

        dataFile.open(fileName, ios::in);
        if (!dataFile)
                cout << "Unable to open " << fileName << ". No Mobile objs added.\n";

        for(;;)
                {
                int i;

                dataFile.getline(tmp, sizeof(tmp));
                i = strlen(tmp);
                if (i <= 0)
                        break;
                cout << "Length of input line is: " << i << "\n";
                if ((tmp[0] != GOAL)
                && (tmp[0] != START))
                        {
//                      cout << "OBJECT line is: " << tmp << "\n";
                        if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                                {
```

```
                              SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
                              SetPointType(domainTimeSlices-1, w, l, MOBILE_OBJECT);
                              newObj = objList.BuildNewObject(w, l);
                              if (newObj)
                                     {
                                     objList.InsertNewObject(newObj);
//                                   cout << "Added obj to obj list\n";
                                     }
                              }
                       else
                              cout << "Improperly formatted line ignored\n";
                       }
              }
       dataFile.close();
       }


void Domain::SetPermObjsInTimeSlice(int timeSlice)
       {
       int w, l;

       for(w=0; w<domainWidth-5; w++)
              {
              l=2;
              switch(w)
                     {
                     case 3:
                     case 4:
                     case 5:
                            break;
                     default:
                            SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                            SetPointType(timeSlice, w, l, OBJECT);
//                          SetAdjObjsInTimeSlice(timeSlice, w, l);
                            break;
                     }
              }
       for(w=0; w<domainWidth-10; w++)
              {
              l=7;
              SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
              SetPointType(timeSlice, w, l, OBJECT);
//            SetAdjObjsInTimeSlice(timeSlice, w, l);
              }
       for(l=0, w=domainWidth-10; l<domainLength-2; l++)
              {
              switch(l)
                     {
                     case 3:
                     case 4:
                     case 5:
                     case 6:
                     case 14:
                     case 15:
                     case 16:
                            break;
                     default:
                            SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                            SetPointType(timeSlice, w, l, OBJECT);
//                          SetAdjObjsInTimeSlice(timeSlice, w, l);
                            break;
                     }
              }
       for(l=2, w=domainWidth-5; l<domainLength-7; l++)
              {
              switch(l)
                     {
                     case 9:
                     case 10:
                     case 11:
                            break;
                     default:
                            SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                            SetPointType(timeSlice, w, l, OBJECT);
//                          SetAdjObjsInTimeSlice(timeSlice, w, l);
                            break;
                     }
              }
       for(w=domainWidth-5, l=domainLength-7; w<domainWidth; w++)
              {
              SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
              SetPointType(timeSlice, w, l, OBJECT);
//            SetAdjObjsInTimeSlice(timeSlice, w, l);
              }
```

```
	for(w=domainWidth-10, l=domainLength-2; w<domainWidth; w++)
		{
		SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
		SetPointType(timeSlice, w, l, OBJECT);
//		SetAdjObjsInTimeSlice(timeSlice, w, l);
		}
	}


void Domain::SetPointFrom(int timeSlice, int width, int length, int from)
	{
	POINT *tmp;

	if ((width < 0)
	|| (length < 0)
	|| (timeSlice < 0)
	|| (width >= domainWidth)
	|| (length >= domainLength)
	|| (timeSlice >= domainTimeSlices))
		return;
	tmp = GetPoint(timeSlice, width, length);
	tmp->from = from;
	}

void Domain::SetPointType(int timeSlice, int width, int length, char type)
	{
	char oldType;
	POINT *tmp;

	if ((width < 0)
	|| (length < 0)
	|| (timeSlice < 0)
	|| (width >= domainWidth)
	|| (length >= domainLength)
	|| (timeSlice >= domainTimeSlices))
		return;
	tmp = GetPoint(timeSlice, width, length);
	oldType = tmp->type;
	tmp->type = type;
	if (oldType != OBJECT)
		{
//		SetAdjObjsInTimeSlice(timeSlice, width, length);
		}
	}


void Domain::SetStartFromFile(char*fileName)
	{
	int w, l;
	char recType;
	char tmp[256];
	fstream dataFile;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Start added.\n";

	for(;;)
		{
		dataFile. getline(tmp, sizeof(tmp));
		if (strlen(tmp) == 0)
			break;
		if (tmp[0] == START)
			{
//			cout << "The START line is: " << tmp << "\n";
			if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
				{
				SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
				SetPointType(domainTimeSlices-1, w, l, START);
				break;
				}
			else
				cout << "Improperly formatted line ignored\n";
			}
		}
	dataFile.close();
	}


void Domain::SetVerticesInTimeSlice(int t)
	{
	int w, l;
	int diff_counter, corner_counter;
	int side_bits, corner_bits;
```

```
	for(w=0; w<domainWidth; w++)
		{
		for(l=0; l<domainLength; l++)
			{
			if (!IsPointClear(t, w, l))
				continue;

			corner_bits = BITMASK_CLEAR;
			corner_counter = 0;
			if (IsPointNearObject(t, w-1, l-1))
				{
				corner_bits |= BITMASK_LEFT;
				corner_bits |= BITMASK_TOP;
				corner_counter++;
				}
			if (IsPointNearObject(t, w-1, l+1))
				{
				corner_bits |= BITMASK_LEFT;
				corner_bits |= BITMASK_BOTTOM;
				corner_counter++;
				}
			if (IsPointNearObject(t, w+1, l+1))
				{
				corner_bits |= BITMASK_RIGHT;
				corner_bits |= BITMASK_BOTTOM;
				corner_counter++;
				}
			if (IsPointNearObject(t, w+1, l-1))
				{
				corner_bits |= BITMASK_RIGHT;
				corner_bits |= BITMASK_TOP;
				corner_counter++;
				}
			if (corner_bits != BITMASK_CLEAR)
				{
				side_bits = BITMASK_CLEAR;
				if (IsPointNearObject(t, w-1, l))
					side_bits |= BITMASK_LEFT;
				if (IsPointNearObject(t, w, l-1))
					side_bits |= BITMASK_TOP;
				if (IsPointNearObject(t, w+1, l))
					side_bits |= BITMASK_RIGHT;
				if (IsPointNearObject(t, w, l+1))
					side_bits |= BITMASK_BOTTOM;

				for(diff_counter=0;
					(side_bits != BITMASK_CLEAR)
					|| (corner_bits != BITMASK_CLEAR);
					side_bits >>= 1, corner_bits >>= 1)
					{
					if ((corner_bits & (int)0x01)
					&& ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
						{
						diff_counter++;
						continue;
						}
					if ((side_bits & (int)0x01)
					&& ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
						{
						diff_counter++;
						continue;
						}
					}
				if ((diff_counter > 2)
				|| ((diff_counter == 2) && (corner_counter == 1)))
					{
					SetPointFrom(t, w, l, FROM_NOWHERE);
					SetPointType(t, w, l, VERTEX);

				}
			}
		}
	}

// edge.hpp
#define DEBUG_FILENAME "DEBUG.LOG"

typedef struct van
   {
	int t,w,l;
	float dist;
	struct van *prev, *next;
	} EDGE_NODE;
```

```
class EdgeList
	{
	public:
		EdgeList();
		~EdgeList();
	EDGE_NODE* BuildNewEdge(int, int, int, float);
		void DelAllEdges(void);
		void DelEdge(EDGE_NODE*);
		void DelEdgeToVertex(int, int, int);
	EDGE_NODE* GetFirstEdge(void);
	EDGE_NODE* GetNextEdge(EDGE_NODE*);
	void InsertNewEdge(EDGE_NODE*);
		void ListAllEdges(int);
   private:
	EDGE_NODE *edgeHead;
	};

// edge.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>
#include <alloc.h>		// for coreleft()
#include <stdlib.h>		// for itoa()
#include <string.h>		// for strcpy()
#include "edge.hpp"


EdgeList::EdgeList()
	{
	edgeHead = (EDGE_NODE*)NULL;
//	cout << "EdgeList Constructor\n";
	}


EdgeList::~EdgeList()
	{
	while (edgeHead != (EDGE_NODE*)NULL)
		DelEdge(edgeHead);
//	cout << "EdgeList Destructor\n";
	}


EDGE_NODE* EdgeList::BuildNewEdge(int t, int w, int l, float dist)
	{
	EDGE_NODE *a;

//	if (otherVertex != (void*)NULL)
//		{
		a = (EDGE_NODE*)farcalloc(1, sizeof(EDGE_NODE));
		if (a == (EDGE_NODE*)NULL)
			{
			cout << "Out of Memory in BuildNewEdge()\n";
	  getch();
			exit(0);
			}
		(*a).t = t;
		(*a).w = w;
		(*a).l = l;
		(*a).dist = dist;
		(*a).prev = (*a).next = (EDGE_NODE*)NULL;
		return(a);
//		}
//	return((EDGE_NODE*)NULL);
	}


void EdgeList::DelAllEdges()
	{
	EDGE_NODE*a;

	for(a = GetFirstEdge();
		a != (EDGE_NODE*)NULL;
		a = GetFirstEdge())
		{
		DelEdge(a);
		}
	}

void EdgeList::DelEdge(EDGE_NODE*a)
	{
	EDGE_NODE*tmp;
```

```
//      cout << "edge 0\n";
        if (a != (EDGE_NODE*)NULL)
                {
                if ((a->prev == (EDGE_NODE*)NULL)  // del last remaining edge
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 1\n";
                        free(a);
                        edgeHead = (EDGE_NODE*)NULL;
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge in middle
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 2\n";
                        tmp = a->prev;
                        tmp->next = a->next;
                        tmp = a->next;
                        tmp->prev = a->prev;
                        free(a);
                        return;
                        }
                if ((a->prev == (EDGE_NODE*)NULL)  // del edge at sol
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 3\n";
                        tmp = a->next;
                        tmp->prev = (EDGE_NODE*)NULL;
                        edgeHead = tmp;
                        free(a);
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge at eol
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 4\n";
                        tmp = a->prev;
                        tmp->next = (EDGE_NODE*)NULL;
                        free(a);
                        return;
                        }
                }
        }


void EdgeList::DelEdgeToVertex(int t, int w, int l)
        {
        EDGE_NODE*tmp;

        for(tmp = GetFirstEdge(); tmp != (EDGE_NODE*)NULL; tmp = GetNextEdge(tmp))
                {
                if ((tmp->t == t)
        && (tmp->w == w)
        && (tmp->l == l))
                        {
                        DelEdge(tmp);
                        break;
                        }
                }
        }


EDGE_NODE* EdgeList::GetFirstEdge()
        {
   return(edgeHead);
   }


EDGE_NODE* EdgeList::GetNextEdge(EDGE_NODE*cur)
   {
   return(cur->next);
   }


void EdgeList::InsertNewEdge(EDGE_NODE *edge)
        {
        EDGE_NODE*cur;

        if (edge == (EDGE_NODE*)NULL)
                return;

        if (edgeHead == (EDGE_NODE*)NULL)
                {
                edgeHead = edge;
```

```
                edge->prev = edge->next = (EDGE_NODE*)NULL;
//              cout << "Inserted edge into empty list ";
//              cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//              cout << "Dist = " << edge->dist << "\n";
                return;
                }

        for(cur=edgeHead; cur != (EDGE_NODE*)NULL; cur=(*cur).next)
                {
                if ( (edge->t > cur->t)
                || ((edge->t == cur->t)
                        && (edge->w > cur->w))
                || ((edge->t == cur->t)
                        && (edge->w == cur->w)
                        && (edge->l > cur->l)) )
                        {                               // insert after cur
                        if (cur->next == (EDGE_NODE*)NULL)
                                {
                                edge->next = (EDGE_NODE*)NULL;// no more so append to eol
                                edge->prev = cur;
                                cur->next = edge;
                                break;
                                }
                        else
                                continue;                       // try next one
                        }
                if ( (edge->t < cur->t)
                || ((edge->t == cur->t)
                        && (edge->w < cur->w))
/*              || ((edge->t < cur->t)*/
                || ((edge->t == cur->t)
                        && (edge->w == cur->w)
                        && (edge->l < cur->l)) )
                        {                               // insert before cur
                        if (cur->prev == (EDGE_NODE*)NULL)
                                {
                                edge->prev = (EDGE_NODE*)NULL;  // at start of list
                                edge->next = cur;
                                cur->prev = edge;
                                edgeHead = edge;
                                break;
                                }
                        else
                                {
                                edge->prev = cur->prev;     // in middle/end of list
                                edge->next = cur;
                                cur->prev = edge;
                                cur = edge->prev;
                                cur->next = edge;
                                break;
                                }
                        }
                if ( (edge->t == cur->t)
                && (edge->w == cur->w)
                && (edge->l == cur->l) )
                        {
                        /* already here - replace it ! */
/*                      cout << "Already here - replacing values and disposing of new edge ";
                        cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
                        cout << "Dist = " << edge->dist << "\n";
                        cur->w = edge->w;
                        cur->l = edge->l;
                        cur->dist = edge->dist;*/
//                      cout << " Already here - ignoring...\n";
                        cout << "!";
                        free(edge);
                        break;
                        }
                }
//      cout << "Inserted edge into list ";
//      cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//      cout << "Dist = " << edge->dist << "\n";
        }


void EdgeList::ListAllEdges(int debugFlag)
        {
        char tmp[256];
        fstream debugFile;
        EDGE_NODE *edge;

        strcpy(tmp, "List of all edges in list\n");
        cout << tmp;
```

```
	if (debugFlag)
		{
		debugFile.open(DEBUG_FILENAME, ios::app);
		debugFile.write(tmp, strlen(tmp));
		debugFile.close();
		}
	for(edge = edgeHead; edge != (EDGE_NODE*)NULL; edge = (*edge).next)
		{
		sprintf(tmp, "Edge to (%d,%d,%d). Dist is %f\n",
					(*edge).t, (*edge).w, (*edge).l, (*edge).dist);
		cout << tmp;
		if (debugFlag)
			{
			debugFile.open(DEBUG_FILENAME, ios::app);
			debugFile.write(tmp, strlen(tmp));
			debugFile.close();
			}
		}
	strcpy(tmp, "-----------------\n");
	cout << tmp;
	if (debugFlag)
		{
		debugFile.open(DEBUG_FILENAME, ios::app);
		debugFile.write(tmp, strlen(tmp));
		debugFile.close();
		}
	}

// object.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define NUM_DIRS 7 /* this is the 8 horizontal directions; (0->7) */
enum directions { front, back, left, right, frontright, frontleft, backright, backleft, up, down, clear};


typedef struct o
	{
	int direction, velocity;
	int w, l;
	struct o *prev, *next;
	} OBJECT_NODE;


class ObjectList
	{
	public:
		ObjectList();
		~ObjectList();
		OBJECT_NODE* BuildNewObject(int, int);
	void DelAllObjects(void);
	void DelObject(OBJECT_NODE*);
	OBJECT_NODE* GetNextObject(void);
		void InsertNewObject(OBJECT_NODE*);
	void ListAllObjects(void);
	private:
		OBJECT_NODE *objectHead;
	};

// object.cpp
#include <iostream.h>
#include <alloc.h>		// for coreleft()
#include <stdlib.h>		// for itoa()
#include <string.h>		// for strcpy()
#include <conio.h>
// #include "edge.hpp"
#include "object.hpp"

ObjectList::ObjectList()
	{
	objectHead = (OBJECT_NODE*)NULL;
	cout << "ObjectList Constructor\n";
	}
```

```
ObjectList::~ObjectList()
        {
        cout << "ObjectList Destructor\n";
        DelAllObjects();
        }


OBJECT_NODE* ObjectList::BuildNewObject(int w, int l)
        {
        OBJECT_NODE *newPtr;

        newPtr = (OBJECT_NODE*)farcalloc(1, sizeof(OBJECT_NODE));
        if (newPtr == (OBJECT_NODE*)NULL)
                {
                cout << "Out of memory in BuildNewObject()\n";
       getch();
                return(NULL);
                }
        (*newPtr).direction = rand() % NUM_DIRS;
        (*newPtr).velocity = 1;
        (*newPtr).prev = (*newPtr).next = (OBJECT_NODE*)NULL;
        (*newPtr).w = w;
        (*newPtr).l = l;
        return(newPtr);
        }


void ObjectList::DelAllObjects()
        {
        OBJECT_NODE *tmp;

        while (objectHead != (OBJECT_NODE*)NULL)
                {
                tmp = objectHead;
                objectHead = (*objectHead).next;
                DelObject(tmp);
                }
        }


void ObjectList::DelObject(OBJECT_NODE *todie)
   {
        OBJECT_NODE *cur;

        if (((*todie).prev != (OBJECT_NODE*)NULL)
        && ((*todie).next != (OBJECT_NODE*)NULL))
                {
//              cout << "Deleted Object (" << todie->w << "," << todie->l << ")\n";
                cur=(*todie).prev;
                if (cur == todie)
                        {
                        objectHead = (OBJECT_NODE*)NULL;
//                      cout << "Object List Empty\n";
                        free(todie);
                        return;
                        }
                else
                        {
                        (*cur).next=(*todie).next;
                        cur = (*todie).next;
                        (*cur).prev = (*todie).prev;
                        if (objectHead == todie)
                                objectHead = (*todie).next;
        free(todie);
          return;
                        }
                }
        cout << "***Did NOT delete rotton Object (" << todie->w << "," << todie->l << ")\n";
        }

OBJECT_NODE* ObjectList::GetNextObject()
   {
   OBJECT_NODE*tmp;

        if (objectHead == (OBJECT_NODE*)NULL)
                return((OBJECT_NODE*)NULL);

        tmp = objectHead;
   objectHead = objectHead->next;
   return(tmp);
```

```
    }


void ObjectList::InsertNewObject(OBJECT_NODE*newPtr)
        {
        OBJECT_NODE *cur;

        if (newPtr == (OBJECT_NODE*)NULL)
                return;

        if (objectHead == (OBJECT_NODE*)NULL)
                {
                objectHead = newPtr;
                (*newPtr).prev = (*newPtr).next = newPtr;
                }
        else
                {
                /* insert before first node */
                cur = objectHead;
                (*newPtr).next = cur;
                (*newPtr).prev = (*cur).prev;
                (*cur).prev = newPtr;

                cur = (*newPtr).prev;
                (*cur).next = newPtr;
                objectHead = newPtr;
        }
//      cout << "Inserted object (" << newPtr->w << "," << newPtr->l << ")\n";
        }


void ObjectList::ListAllObjects()
        {
    OBJECT_NODE*tmp;

    if (objectHead == (OBJECT_NODE*)NULL)
        {
        cout << "ObjectList is empty\n";
        return;
        }
    for(tmp=objectHead;;tmp = tmp->next)
        {
         cout << "Object:(" << tmp->w << "," << tmp->l << ")\n";
        tmp = tmp->next;
        if (tmp == objectHead)
           {
           cout << "ObjectList ended\n";
           break;
           }
        }
         }

// vertex.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00

#define DEBUG_FILENAME "DEBUG.LOG"

typedef struct vn
        {
        int t,w,l;
        char nodeType;
        struct vn *searchPrev, *searchNext;
        float searchDist;
        int searchMarker;
        EdgeList *edgeList;
        struct vn *pathFrom, *pathTo;
        struct vn *prev, *next;
        } VERTEX_NODE;
```

```
class VertexList
        {
        public:
                VertexList();
                ~VertexList();
                void AddToSearchList(VERTEX_NODE*);
        VERTEX_NODE* BuildNewVertex(int,int,int,char);
                int CalcRobotDir(VERTEX_NODE*);
                void DelAllVertices(void);
        void DelVertex(VERTEX_NODE*);
        int FindPath(void);
        VERTEX_NODE* FindVertex(int, int, int);
                VERTEX_NODE* GetFirstVertex(void);
                VERTEX_NODE* GetGoalVertex(void);
                VERTEX_NODE* GetNextVertex(VERTEX_NODE*);
                int GetSearchMarker(void);
        VERTEX_NODE* GetStartVertex(void);
        void InsertAllVertices(void);
                void InsertNewVertex(VERTEX_NODE*);
                void ListAllVertices(int);
        void ListSearchList(int);
//              void MoveRobot(int, int, int, int);
                void MarkPath(VERTEX_NODE*);
                void RemoveFromSearchList(VERTEX_NODE*);
        void RemoveVertex(VERTEX_NODE*);
                void TrimSearchList(void);
        private:
                VERTEX_NODE* vertexHead;
        VERTEX_NODE* searchHead;
        int searchMarker;
        float searchTrimDist;
        };

//vertex.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>              // for getch()
#include <alloc.h>              // for coreleft()
#include <stdlib.h>             // for itoa()
#include <string.h>             // for strcpy()
#include "edge.hpp"
#include "vertex.hpp"

VertexList::VertexList()
        {
        vertexHead = (VERTEX_NODE*)NULL;
   searchHead = (VERTEX_NODE*)NULL;
   searchMarker = 0;
   searchTrimDist = -1.0;
   cout << "Initialised Vertex class\n";
        }


VertexList::~VertexList()
        {
        cout << "VertexList destructor started\n";
        DelAllVertices();
        cout << "VertexList destructor ended!\n";
        }


void VertexList::AddToSearchList(VERTEX_NODE*a)
        {
        VERTEX_NODE *tmp;

        if (a == (VERTEX_NODE*)NULL)
                return;

        if (((*a).searchPrev != (VERTEX_NODE*)NULL) // if already in fringe
        || ((*a).searchNext != (VERTEX_NODE*)NULL)) // list, dont add again
                return;

        if (searchHead == (VERTEX_NODE*)NULL)
                {
                searchHead = a;
                a->pathFrom = a->pathTo = (VERTEX_NODE*)NULL;
                a->searchPrev = a->searchNext = a;

//              cout << "Inserted into empty search list (";
//              cout << a->t << "," << a->w << "," << a->l << ")\n";
```

```
			}
	else
			{
			a->searchNext = searchHead;
			a->searchPrev = searchHead->searchPrev;
			searchHead->searchPrev = a;
			tmp = a->searchPrev;
			tmp->searchNext = a;
//			cout << "Appended to search list (";
//			cout << a->t << "," << a->w << "," << a->l << ")\n";
			}
	}


VERTEX_NODE* VertexList::BuildNewVertex(int t, int w, int l, char nodeType)
	{
	VERTEX_NODE *newPtr;

	newPtr = (VERTEX_NODE*)farcalloc(1, sizeof(VERTEX_NODE));
	if (newPtr == (VERTEX_NODE*)NULL)
			{
			cout << "Farcalloc() failed in BuildNewVertex!\n";
	getch();
			exit(0);
			}
	(*newPtr).t = t;
	(*newPtr).w = w;
	(*newPtr).l = l;
	(*newPtr).nodeType = nodeType;
	(*newPtr).edgeList = new EdgeList();
	(*newPtr).pathTo = (*newPtr).pathFrom = (VERTEX_NODE*)NULL;
	(*newPtr).searchDist = 0.0;
	(*newPtr).searchPrev = (*newPtr).searchNext = (VERTEX_NODE*)NULL;
	(*newPtr).prev = (*newPtr).next = (VERTEX_NODE*)NULL;
	if (nodeType == START)
			{
			(*newPtr).searchMarker = searchMarker+1;
			AddToSearchList(newPtr);
			}
	else
			(*newPtr).searchMarker = searchMarker;
	return(newPtr);
	}


void VertexList::DelAllVertices(
	{
	while (vertexHead != (VERTEX_NODE*)NULL)
			{
//			cout << "About to delete (" << vertexHead->t << "," << vertexHead->w ;
//			cout << "," << vertexHead->l << ") at addr: " << vertexHead ;
//			cout << ": Prev:" << vertexHead->prev << ": Next:" << vertexHead->next <<"\n";
			DelVertex(vertexHead);
			}
	cout << "----------\n";
//	getch();
	}


void VertexList::DelVertex(VERTEX_NODE *todie)
	{
	VERTEX_NODE *tmp;
	EDGE_NODE*a;

	if (todie == (VERTEX_NODE*)NULL)
			return;

	for(a = todie->edgeList->GetFirstEdge();
			a != (EDGE_NODE*)NULL;
			a = todie->edgeList->GetFirstEdge())
			{
	todie->edgeList->DelEdgeToVertex(a->t, a->w, a->l);
	tmp = FindVertex(a->t, a->w, a->l);
	if (tmp != (VERTEX_NODE*)NULL)
		{
		tmp->edgeList->DelEdgeToVertex(todie->t, todie->w, todie->l);
		}
//			tmp = (VERTEX_NODE*)(a->otherVertex);
//			cout << "On vertex (" << todie->t << "," << todie->w << ",";
//			cout << todie->l << "):Del edge to (" << tmp->t << "," << tmp->w << ",";
//			cout << tmp->l << ")\n";
//			cout << "todie at:" << todie << ". a at:" << a << "\n";
//			cout << "  tmp at:" << tmp << "\n";
```

```
//              todie->edgeList->DelEdgeToVertex((void*)tmp);
//              cout << "On vertex (" << tmp->t << "," << tmp->w << ",";
//              cout << tmp->l << "):Del edge to (" << todie->t << "," << todie->w << ",";
//              cout << todie->l << ")\n";
//              tmp->edgeList->DelEdgeToVertex((void*)todie);
                }
        delete todie->edgeList;
        RemoveFromSearchList(todie);

        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
                {
                vertexHead = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
                tmp = (*todie).prev;
                (*tmp).next = (*todie).next;
                tmp = (*todie).next;
                (*tmp).prev = (*todie).prev;
                free(todie);
                return;
                }
        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
                vertexHead = tmp = (*todie).next;
                (*tmp).prev = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
                {
                tmp = (*todie).prev;
                (*tmp).next = (VERTEX_NODE*)NULL;
                free(todie);
                return;
                }
        }


int VertexList::FindPath()
   {
        EDGE_NODE *e;
        int goalFound=FALSE;
        float dist;
        VERTEX_NODE *cur, *adj, *t1, *t2;

        if (searchHead->nodeType == START)
                searchMarker = searchHead->searchMarker;
        else
                return(FALSE);

        t1 = GetStartVertex();
        t2 = GetGoalVertex();
        if ((t1 != (VERTEX_NODE*)NULL)
        && (t2 != (VERTEX_NODE*)NULL)
        && (t1->w == t2->w)
        && (t1->l == t2->l))
                {
                t1->pathTo = t2;
                return(TRUE);
                }

        for (cur = searchHead; cur != (VERTEX_NODE*)NULL; cur = searchHead)
                {
                if ((cur->searchDist >= searchTrimDist)
                && (searchTrimDist > 0.0))
                        {
                        RemoveFromSearchList(cur);
                        continue;
                        }
                for(e = cur->edgeList->GetFirstEdge();
                        e != (EDGE_NODE*)NULL;
                        e = cur->edgeList->GetNextEdge(e))
                        {
          adj = FindVertex(e->t, e->w, e->l);
          if (adj == (VERTEX_NODE*)NULL)
             continue;
```

```
//          adj = (VERTEX_NODE*) e->otherVertex;
                    dist = cur->searchDist + e->dist;
                    if ( (adj->searchMarker != searchMarker)
                    || ((adj->searchMarker == searchMarker)
                         && (adj->searchDist > dist)) )
                         {
                         if ((dist <= searchTrimDist)
                         || (searchTrimDist <= 0.0))
                              {
                              adj->pathFrom = cur;
                              adj->searchMarker = searchMarker;
                              adj->searchDist = dist;
                              AddToSearchList(adj);
                              if (adj->nodeType == GOAL)
                                   {
                                   goalFound = TRUE;
                                   searchTrimDist = dist;
                                   MarkPath(adj);
                                   TrimSearchList();
                                   }

                         }
                    }
          RemoveFromSearchList(cur);
          }
     return(goalFound);
     }


VERTEX_NODE* VertexList::FindVertex(int t, int w, int l)
   {
   VERTEX_NODE* cur;

   for(cur = GetFirstVertex();
      cur != (VERTEX_NODE*)NULL;
      cur = GetNextVertex(cur))
      {
      if ((cur->t == t)
      && (cur->w == w)
      && (cur->l == l))
         return(cur);          // found it

      if ((cur->t >= t)
      && (cur->w >= w)
      && (cur->l >= l))
         break;                // passed it - it's not in the list
         }
   return((VERTEX_NODE*)NULL);
   }


VERTEX_NODE* VertexList::GetFirstVertex()
   {
   return(vertexHead);
   }


VERTEX_NODE* VertexList::GetGoalVertex()
   {
   VERTEX_NODE*tmp;

   for(tmp = GetFirstVertex();
      tmp != (VERTEX_NODE*)NULL;
      tmp = GetNextVertex(tmp))
      {
            if (tmp->nodeType == GOAL)
         break;
      }
   return(tmp);
   }


VERTEX_NODE* VertexList::GetNextVertex(VERTEX_NODE*cur)
   {
   return(cur->next);
   }


int VertexList::GetSearchMarker()
     {
     return(searchMarker);
     }
```

```
VERTEX_NODE* VertexList::GetStartVertex()
       {
       VERTEX_NODE*tmp;

       for(tmp = GetFirstVertex();
              tmp != (VERTEX_NODE*)NULL;
              tmp = GetNextVertex(tmp))
              {
              if (tmp->nodeType == START)
         break;
      }
   return(tmp);
   }


void VertexList::InsertNewVertex(VERTEX_NODE*a)
       {
       VERTEX_NODE *cur;

       if (a == (VERTEX_NODE*)NULL)
              return;

       if (vertexHead == (VERTEX_NODE*)NULL)
              {
              vertexHead = a;
              (*a).prev = (*a).next = (VERTEX_NODE*)NULL;
              return;
              }

       for(cur=vertexHead; cur != (VERTEX_NODE*)NULL;   cur=(*cur).next)
              {
              if ( (a->t > cur->t)
      || ((a->t == cur->t)
        && (a->w > cur->w))
       || ((a->t == cur->t)
        && (a->w == cur->w)
              && (a->l > cur->l)) )
                     {                                 // insert after cur
                     if ((*cur).next == (VERTEX_NODE*)NULL)
                            {
                            (*a).next = (*cur).next;    // eol - append new node
                            (*a).prev = cur;
                            (*cur).next = a;
                            break;
                            }
                     else
                            continue;                   // get next node
                     }
              if ( (a->t < cur->t)
              || ((a->t == cur->t)
                     && (a->w < cur->w))
              || ((a->t == cur->t)
                     && (a->w == cur->w)
                     && (a->l < cur->l)) )
                     {                                  // insert before cur
                     if (cur->prev == (VERTEX_NODE*)NULL)
                            {
                            a->prev = (VERTEX_NODE*)NULL;// at start of list
                            a->next = cur;
                            cur->prev = a;
                            vertexHead = a;
                            break;
                            }
                     else
                            {
                            a->prev = cur->prev;         // in middle/end of list
                            a->next = cur;
                            cur->prev = a;
                            cur = a->prev;
                            cur->next = a;
                            break;
                            }
                     }
              if ((a->t == cur->t)
              && (a->w == cur->w)
              && (a->l == cur->l))
                     {                                  // insert after cur at eol
                     if ((*cur).next == (VERTEX_NODE*)NULL)
                            {
                            (*a).next = (VERTEX_NODE*)NULL;
                            (*a).prev = cur;
                            (*cur).next = a;
                            break;
```

```
                        }
                    else
                        continue;
                    }
                }
//      AddToSearchList(a);
//      cout << "Inserted vertex (";
// cout << (*a).t << "," << (*a).w << "," << (*a).l << ")\n";
        }


void VertexList::ListAllVertices(int debugFlag)
        {
        char tmp[256];
        fstream debugFile;
        VERTEX_NODE *cur;

        strcpy(tmp, "List of all nodes in list\n");
        cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        for(cur=vertexHead; cur != (VERTEX_NODE*)NULL; cur=cur->next)
                {
                sprintf(tmp, "Vertex:%c: at (%d,%d,%d)\n",
                                (*cur).nodeType, (*cur).t, (*cur).w, (*cur).l);
                cout << tmp;
                if (debugFlag)
                        {
                        debugFile.open(DEBUG_FILENAME, ios::app);
                        debugFile.write(tmp, strlen(tmp));
                        debugFile.close();
                        }
                cur->edgeList->ListAllEdges(debugFlag);
                }
        }


void VertexList::ListSearchList(int debugFlag)
        {
        char tmp[256];
        fstream debugFile;
        VERTEX_NODE*cur;

        if (searchHead == (VERTEX_NODE*)NULL)
                {
                sprintf(tmp,"Empty SearchList! (marker=%d)\n", searchMarker);
      cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
                return;
                }

        sprintf(tmp, "List of nodes in Search list (marker=%d)\n", searchMarker);
   cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        cur = searchHead;
        do
                {
                sprintf(tmp, "SearchList node (%d,%d,%d) Dist = %f",
              cur->t, cur->w, cur->l, cur->searchDist);
      cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
                cur=cur->searchNext;
                }
        while (cur != searchHead);
   strcpy(tmp, "End of SearchList\n");
```

```
    cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        }


void VertexList::MarkPath(VERTEX_NODE*v)
        {
        VERTEX_NODE*tmp;

        if (v == (VERTEX_NODE*)NULL)
                return;

//      cout << "Path from Goal to Start\n";
        v->pathTo = (VERTEX_NODE*)NULL;
        do
                {
                tmp = v->pathFrom;
//              cout << "(" << v->t << "," << v->w << "," << v->l << ")\n";
                if (tmp != (VERTEX_NODE*)NULL)
                        {
                        tmp->pathTo = v;
                        v = v->pathFrom;
                        }
                }
        while (tmp != (VERTEX_NODE*)NULL);
        }


void VertexList::RemoveFromSearchList(VERTEX_NODE*a)
        {
        VERTEX_NODE *cur;

        if ((a->searchPrev == (VERTEX_NODE*)NULL)
        || (a->searchNext == (VERTEX_NODE*)NULL))
       {
       if (searchHead == a)
          searchHead = (VERTEX_NODE*)NULL;
       return;
       }

        cur=a->searchPrev;
        if (cur == a)
                {
       searchHead = (VERTEX_NODE*)NULL;
//              cout << "Removed Search Node (";
//     cout << a->t << "," << a->w << "," << a->l << "). SearchList empty\n";
                }
        else
                {
                cur->searchNext = a->searchNext;
                cur = a->searchNext;
                cur->searchPrev = a->searchPrev;
                if (searchHead == a)
                        searchHead = a->searchNext;
//              cout << "Removed Search Node (";
//     cout << a->t << "," << a->w << "," << a->l << ").\n";
                }
        a->searchPrev = a->searchNext = (VERTEX_NODE*)NULL;
        }


void VertexList::RemoveVertex(VERTEX_NODE *todie)
        {
        VERTEX_NODE *tmp;

        if (todie == (VERTEX_NODE*)NULL)
                return;

        RemoveFromSearchList(todie);

        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
                {
                vertexHead = (VERTEX_NODE*)NULL;
                return;
                }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
```

```
			tmp = (*todie).prev;
			(*tmp).next = (*todie).next;
			tmp = (*todie).next;
			(*tmp).prev = (*todie).prev;
			return;
			}
		if ( ((*todie).prev == (VERTEX_NODE*)NULL)
		&& ((*todie).next != (VERTEX_NODE*)NULL) )
			{
			vertexHead = tmp = (*todie).next;
			(*tmp).prev = (VERTEX_NODE*)NULL;
			return;
			}
		if ( ((*todie).prev != (VERTEX_NODE*)NULL)
		&& ((*todie).next == (VERTEX_NODE*)NULL) )
			{
			tmp = (*todie).prev;
			(*tmp).next = (VERTEX_NODE*)NULL;
			return;
			}
		}

void VertexList::TrimSearchList()
	{
		VERTEX_NODE *cur, *tmpPtr=(VERTEX_NODE*)0;
		int trimmed=TRUE;

//		cout << "Trimming search list to " << searchTrimDist << " or less\n";
		if ((searchHead == (VERTEX_NODE*)NULL)
	|| (searchTrimDist < 0.0))
			return;

	cur = searchHead;
		while ((tmpPtr != cur) || (trimmed == TRUE))
			{
			if (trimmed==TRUE)
				{
				tmpPtr=cur;
				trimmed=FALSE;
				}
			if ((*cur).searchDist >= searchTrimDist)
				{
//				cout << "Trimmed out (";
//		cout << cur->t << "," << cur->w << "," << cur->l << ") ";
//		cout << "Dist was " << cur->searchDist << "\n";
				if (cur == cur->searchNext)
					{
		RemoveFromSearchList(cur);
					break;
					}
				else
					{
					tmpPtr = cur->searchNext;
					RemoveFromSearchList(cur);
					cur = tmpPtr;
					trimmed = TRUE;
					}
				}
			else
				cur = cur->searchNext;
			}
//	cout << "-------\n";
	}
```

B.8. Partial Update Depth First Graph Theory

The Partial Update Depth First Search Algorithm for Graph Theory which was developed and implemented as part of this project was coded in ANSI C++. The filenames for each of the separate source code files were supplied inside C++ format comments (i.e. //) at the beginning of each file listing. A detailed explanation of the design behind this program was presented in Chapter 4.

```
// main.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
```

```
#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <sys\timeb.h>
#include <dir.h>
#include <ctype.h>
#include "edge.hpp"
#include "vertex.hpp"
#include "object.hpp"
#include "domain.hpp"
#include "bench.hpp"

#define MAPFILE_MASK "MAP*.DAT"

int main(int, char**);
void BuildUpdateListFromDomain(VertexList&, VertexList&, Domain&);
void BuildVertexListFromDomain(VertexList&, Domain&);
void FindAllEdges(VertexList&, Domain&);
void FindEdgesForUpdateNode(VERTEX_NODE*, VertexList&, VertexList&, Domain&);
int MemStatus(char*);
void MergeLists(VertexList&, VertexList&, Domain&);
int SetWorkingDir(void);
void RenameMapFile(char*);


int main(int argc, char** argv)
	{
	char* shortName;
	struct ffblk mapfile;

	for(;;)
		{
		shortName = strstr(argv[0], "\\");
		if (shortName == (char*)NULL)
			{
			shortName = argv[0];
			cout << "This is " << shortName << "\n";
			break;
			}
		else
			argv[0] = shortName+1;
		}

	MemStatus("Free memory before mainloop in main: ");

	if (!SetWorkingDir())
		{
		cout << "Program exiting gracefully\n";
		return(1);
		}
	if (findfirst(MAPFILE_MASK, &mapfile, 0))
		{
		cout << "Program exiting gracefully - no map files found\n";
		return(1);
		}
	do
		{
		char tmp[256];
		int i;
		Benchmark stopWatch(shortName);

		sprintf(tmp, "Starting on %s: ", mapfile.ff_name);
		MemStatus(tmp);
		for (i=0; i<NUMTIMES; i++)
			{
			int timeTaken;
			float distTravelled;
			Domain theWorld(10, 20, 20, mapfile.ff_name);
			VertexList vertList, updateList;

			timeTaken = 0;
			distTravelled = 0.0;
			stopWatch.IterStart(i, mapfile.ff_name);
			stopWatch.Click();
			BuildVertexListFromDomain(vertList, theWorld);
			FindAllEdges(vertList, theWorld);
			for(;;)
				{
				int pathFoundFlag, goalFoundFlag, a, b, c;
				float dist;
				VERTEX_NODE *t1;

				stopWatch.Click();
```

```
                                    theWorld.DrawDomain();
                                    stopWatch.Click();
                                    pathFoundFlag = vertList.FindPath();
                                    stopWatch.Click();

                                    if (pathFoundFlag == TRUE)
                                          {
                                          cout << "Path found to the GOAL!";
                                          cout << "(dist=" << distTravelled << ")\n";
                                          }
                                    else
                                          {
                                          cout << "No path found to the GOAL!";
                                          cout << "(dist=" << distTravelled << ")\n";
                                          }

                                    t1 = vertList.GetStartVertex();
                                    if (pathFoundFlag)
                                          {
                                          a = t1->pathTo->t;
                                          b = t1->pathTo->w;
                                          c = t1->pathTo->l;
                                          }
                                    else
                                          {
                                          a = (t1->t)+1;
                                          b = t1->w;
                                          c = t1->l;
                                          }
                                    goalFoundFlag = theWorld.MoveRobot(t1->t, t1->w, t1->l,

a, b, c, &dist);
                                    distTravelled += dist;
                                    timeTaken++;
                                    if (goalFoundFlag)
                                          {
                                          cout << "MADE IT TO THE GOAL!\n";
                                          break;
                                          }

                                    theWorld.AdvanceTime();
                                    stopWatch.Click();
                                    BuildUpdateListFromDomain(vertList, updateList, theWorld);
                                    MergeLists(vertList, updateList, theWorld);
                                    }
                              stopWatch.IterStop(timeTaken, distTravelled);
                              }
                  stopWatch.LogCalcs();
                  RenameMapFile(mapfile.ff_name);
                  }
      while (!findnext(&mapfile));

      MemStatus("Free memory after mainloop in main: ");
      return(0);
      }


void BuildUpdateListFromDomain(VertexList &vList, VertexList &uList, Domain &domain)
      {
      char type;
      int t=domain.GetDomainTimeSlices(), a;
      int w=domain.GetDomainWidth(), b;
      int l=domain.GetDomainLength(), c;
      VERTEX_NODE*newPtr;

      for(a=0; a<t; a++)
        {
            for(b=0; b<w; b++)
                  {
                  for(c=0; c<l; c++)
                        {
                        type = domain.GetPointType(a, b, c);
                        if ((type == VERTEX)
                        || (type == START)
                        || (type == GOAL))
                              {
                              VERTEX_NODE *tmpv, *tmpu;

                              tmpv = vList.FindVertex(a, b, c);
                              tmpu = uList.FindVertex(a, b, c);

                              if ((!tmpv)
                              && (!tmpu))
```

```
                                        {
                                        newPtr = uList.BuildNewVertex(a, b, c, type);
                                        uList.InsertNewVertex(newPtr);
                                        continue;
                                        }

                                if ((tmpv != (VERTEX_NODE*)NULL)
                                && (tmpv->nodeType != type))
                                        {
                                        vList.RemoveVertex(tmpv);
                                        vList.DelVertex(tmpv);
                                        newPtr = uList.BuildNewVertex(a, b, c, type);
                                        uList.InsertNewVertex(newPtr);
                                        continue;
                                        }

                                if ((tmpu != (VERTEX_NODE*)NULL)
                                && (tmpu->nodeType != type))
                                        {
                                        uList.RemoveVertex(tmpu);
                                        uList.DelVertex(tmpu);
                                        newPtr = uList.BuildNewVertex(a, b, c, type);
                                        uList.InsertNewVertex(newPtr);
                                        continue;
                                        }
                                }
                        else
                                {
                                VERTEX_NODE *v;

                                v = vList.FindVertex(a, b, c);
                                if (v!=(VERTEX_NODE*)NULL)
                                        {
                                        EDGE_NODE*e;

                                        for(e = v->edgeList->GetFirstEdge();
                                                e != (EDGE_NODE*)NULL;
                                                e = v->edgeList->GetNextEdge(e))
                                                {
                                                VERTEX_NODE* v2;

                                                v2 = vList.FindVertex(e->t, e->w, e->l);
                                                if (v2 != (VERTEX_NODE*)NULL)
                                                        {
                                                        vList.RemoveVertex(v2);
                                                        uList.InsertNewVertex(v2);
                                                        v2->edgeList->DelAllEdges();
                                                        }
                                                }
                                        vList.DelVertex(v);
                                        }
                                v = uList.FindVertex(a, b, c);
                                if (v != (VERTEX_NODE*)NULL)
                                        {
                                        uList.DelVertex(v);
                                        }

                        }
                }
        }
}


void BuildVertexListFromDomain(VertexList &vList, Domain &domain)
   {
        char type;
        int t=domain.GetDomainTimeSlices(), a;
        int w=domain.GetDomainWidth(), b;
        int l=domain.GetDomainLength(), c;
        VERTEX_NODE*newPtr;

        for(a=0; a<t; a++)
          {
                for(b=0; b<w; b++)
                        {
                        for(c=0; c<l; c++)
                                {
                                type = domain.GetPointType(a, b, c);
                                if ((type == VERTEX)
                                || (type == START)
                                || (type == GOAL))
                                        {
                                newPtr = vList.BuildNewVertex(a, b, c, type);
                                vList.InsertNewVertex(newPtr);
```

```
                        }
                    }
                }
            }

void FindAllEdges(VertexList &vList, Domain &domain)
        {
        EDGE_NODE*e;
        VERTEX_NODE*a,*b;
        float dist;

        for(a = vList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = vList.GetNextVertex(a))
                {
                for(b = vList.GetNextVertex(a);
                        b != (VERTEX_NODE*)NULL;
                        b = vList.GetNextVertex(b))
          {
                        if (a == b)
                                continue;

                        if ((a->t == b->t)
                        && (a->w == b->w)
                        && (a->l == b->l))
                                continue;

          dist = domain.CheckLine(a->t, a->w, a->l, b->t, b->w, b->l);
          if (dist > 0.0)
              {
                                e = a->edgeList->BuildNewEdge(b->t, b->w, b->l, dist);
                                a->edgeList->InsertNewEdge(e);
                                }
                        dist = domain.CheckLine(b->t, b->w, b->l, a->t, a->w, a->l);
                        if (dist > 0.0)
                                {
                                e = b->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                                b->edgeList->InsertNewEdge(e);
                                }
                        }
                }
        }

void FindEdgesForUpdateNode(VERTEX_NODE*cur, VertexList &vList, VertexList &uList, Domain &domain)
        {
        EDGE_NODE*e;
        VERTEX_NODE*a;
        float dist;

        for(a = vList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = vList.GetNextVertex(a))
                {
       if (a == cur)
                continue;

                if ((a->t == cur->t)
                && (a->w == cur->w)
                && (a->l == cur->l))
                        continue;

       dist = domain.CheckLine(a->t, a->w, a->l, cur->t, cur->w, cur->l);
       if (dist > 0.0)
           {
                        e = a->edgeList->BuildNewEdge(cur->t, cur->w, cur->l, dist);
                        a->edgeList->InsertNewEdge(e);
                        }
                dist = domain.CheckLine(cur->t, cur->w, cur->l, a->t, a->w, a->l);
                if (dist > 0.0)
                        {
                        e = cur->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                        cur->edgeList->InsertNewEdge(e);
                        }
                }

        for(a = uList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = uList.GetNextVertex(a))
                {
       if (a == cur)
                continue;
```

```
            if ((a->t == cur->t)
            && (a->w == cur->w)
            && (a->l == cur->l))
                    continue;

        dist = domain.CheckLine(a->t, a->w, a->l, cur->t, cur->w, cur->l);
        if (dist > 0.0)
                    {
                    e = a->edgeList->BuildNewEdge(cur->t, cur->w, cur->l, dist);
                    a->edgeList->InsertNewEdge(e);
                    }
            dist = domain.CheckLine(cur->t, cur->w, cur->l, a->t, a->w, a->l);
            if (dist > 0.0)
                    {
                    e = cur->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                    cur->edgeList->InsertNewEdge(e);
          }
            }
        }


int MemStatus(char *StatusMessage)
        {
        char tmp[256];
        fstream debugFile;
        long MemLeft;
        int ret;

        debugFile.open("DEBUG.LOG", ios::app);
        if (!debugFile)
                cout << "Unable to open DEBUG.LOG\n";

        MemLeft = (long) coreleft();
        sprintf(tmp, "%s%ld\n", StatusMessage, MemLeft);
        debugFile.write(tmp, strlen(tmp));
        debugFile.close();
//      cout << StatusMessage << MemLeft << "\n";

        ret = farheapcheck();
        if (ret == _HEAPOK)
                cout << "Heap ok" << StatusMessage << "\n";
        else
                {
                cout << "Heap error <" << ret << ">" << StatusMessage<< "\n";
                getch();
                return(FALSE);
                }
        return(TRUE);
        }


void MergeLists(VertexList &vList, VertexList &uList, Domain &domain)
    {
    VERTEX_NODE* cur;

    for (cur=uList.GetFirstVertex();
       cur!=(VERTEX_NODE*)NULL;
       cur=uList.GetFirstVertex())
       {
       uList.RemoveVertex(cur);
                FindEdgesForUpdateNode(cur, vList, uList, domain);
                vList.InsertNewVertex(cur);
                }
        cur = vList.GetStartVertex();
        if (cur != (VERTEX_NODE*)NULL)
                {
                cur->searchMarker = vList.GetSearchMarker() + 1;
                vList.AddToSearchList(cur);
                }
        }


int SetWorkingDir()
        {
        char mapdir[256];

        cout <<"Enter the directory containing map files, or \"q\" for quit:";
        cin >> mapdir;
        if (toupper(mapdir[0]) == 'Q')
                {
                cout << "Quitting...\n";
```

```
                return(FALSE);
                }
        if (chdir(mapdir))
                {
                cout << "The directory " << mapdir << " could not be found.\n";
                return(FALSE);
                }
        cout << "Made " << mapdir << " the current directory.\n";
        return(TRUE);
        }


void RenameMapFile(char*filename)
        {
        char newfilename[128];
        char* ch;
        int i=(int)'.';

        strcpy(newfilename, filename);
        ch = strrchr(newfilename,i);
        if (ch != (char*)NULL)
                {
                strcpy(ch, ".bak");
                rename(filename, newfilename);
                }
        else
                exit(1);
        }

// bench.hpp
#define FALSE 0
#define TRUE !FALSE
#define NUMTIMES 10
#define MAXFILENAME 13

class Benchmark
        {
        public:
                Benchmark(char*);
                ~Benchmark();
                void Click(void);
                void IterStart(int, char*);
                void IterStop(int, float);
                void LogCalcs(void);
        private:
                void Diff(struct timeb*, struct timeb*, struct timeb*);

                fstream logFile;
                fstream avgFile;
                char mapFileName[MAXFILENAME];
                struct timeb benchmarks[NUMTIMES][2];
                float distRobotTravelled[NUMTIMES];
                int timeTaken[NUMTIMES];
                int currentIter;
                int clickToggleFlag;
                struct timeb elapsedTime, computeTime;
                struct timeb startTime, clickOnTime, clickOffTime;
        };

// bench.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <sys\timeb.h>
#include "bench.hpp"


Benchmark::Benchmark(char *fileName)
        {
        char *tmp;
        char logname[MAXFILENAME];

        while((tmp = strchr(fileName, '\\')) != (char*)NULL)
                fileName = tmp+1;
        strcpy(logname, fileName);
        if ((tmp = strchr(logname, '.')) != (char*)NULL)
                strcpy(tmp, ".LOG");
        else
                strcat(logname, ".LOG");
        logFile.open(logname, ios::app);
        if (!logFile)
```

```
		cout << "Unable to open " << logname << "\n";

	if ((tmp = strchr(logname, '.')) != (char*)NULL)
		strcpy(tmp, ".AVG");
	else
		strcat(logname, ".AVG");
	avgFile.open(logname, ios::app);
	if (!avgFile)
		cout << "Unable to open " << logname << "\n";

	cout << "Initialised Benchmark class\n";
	}


Benchmark::~Benchmark()
	{
	if (logFile)
		{
		logFile.flush();
		logFile.close();
		}
	if (avgFile)
		{
		avgFile.flush();
		avgFile.close();
		}
	cout << "Closed log files and Destroying Benchmark class\n";
	}


void Benchmark::Click()
	{
	switch(clickToggleFlag)
		{
		case TRUE:
			ftime(&clickOnTime);
			clickToggleFlag = FALSE;
			break;
		case FALSE:
		default:
			ftime(&clickOffTime);
			Diff(&clickOnTime, &clickOffTime, &computeTime);
			clickToggleFlag = TRUE;
		}
	}


void Benchmark::Diff(struct timeb*start, struct timeb*stop, struct timeb*diff)
	{
	if ((*stop).millitm < (*start).millitm)
		{
		(*stop).millitm += (short)1000; /* carry when subtracting, stops*/
		(*start).time += 1L;            /* negative wraparound problems!*/
		}
	(*diff).millitm += (*stop).millitm - (*start).millitm;
	(*diff).time += (long)((*diff).millitm / (short)1000);
	(*diff).millitm %= (short)1000;
	(*diff).time += ((*stop).time-(*start).time);
	}


void Benchmark::IterStart(int i, char*s)
	{
	currentIter = i;
	clickToggleFlag = TRUE;
	computeTime.time = elapsedTime.time = 0L;
	computeTime.millitm = elapsedTime.millitm = 0;

	strcpy(mapFileName, s);

	ftime(&startTime);
	}


void Benchmark::IterStop(int t, float distTravelled)
	{
	struct timeb stopTime;

	ftime(&stopTime);
	Diff(&startTime, &stopTime, &elapsedTime);

	benchmarks[currentIter][0].time = elapsedTime.time;
	benchmarks[currentIter][0].millitm = elapsedTime.millitm;
	benchmarks[currentIter][1].time = computeTime.time;
```

```
        benchmarks[currentIter][1].millitm = computeTime.millitm;
        timeTaken[currentIter] = t;
        distRobotTravelled[currentIter] = distTravelled;
        }



void Benchmark::LogCalcs()
        {
        char tmp[256];
        float avgDist;
        struct timeb avg;
        int i, avgTimeTaken;

        for(i=0, avg.time=0L, avg.millitm=0, avgDist=0.0, avgTimeTaken=0;
                i<NUMTIMES;
                i++)
                {
                sprintf(tmp, "%s (%02d) Elapsed time:%05ld.%03d\n", mapFileName, i,
                                        benchmarks[i][0].time, benchmarks[i][0].millitm);
                logFile.write(tmp, strlen(tmp));
                sprintf(tmp, "%s (%02d) Compute time:%05ld.%03d\n", mapFileName, i,
                                        benchmarks[i][1].time, benchmarks[i][1].millitm);
                logFile.write(tmp, strlen(tmp));
                sprintf(tmp, "%s (%02d) Dist travelled:%f\n", mapFileName, i,
                                        distRobotTravelled[i]);
                logFile.write(tmp, strlen(tmp));
                sprintf(tmp, "%s (%02d) Time Slices Taken:%d\n", mapFileName, i,
                                        timeTaken[i]);
                logFile.write(tmp, strlen(tmp));
                avg.time+=benchmarks[i][0].time;
                avg.millitm+=benchmarks[i][0].millitm;
                if (avg.millitm % 1000 != avg.millitm)
                        {
                        avg.time += (long)(avg.millitm / (short)1000);
                        avg.millitm %= (short)1000;
                        }
                }

        sprintf(tmp, "%s Tot. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));
        cout << avg.time << "." << avg.millitm << "\n";
        cout << tmp << "\n";

        i = (int) (avg.time % (long)NUMTIMES);
        avg.time /= (long)NUMTIMES;
        avg.millitm += i * 1000;
        avg.millitm /= NUMTIMES;

        sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));

        for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
                {
                avg.time+=benchmarks[i][1].time;
                avg.millitm+=benchmarks[i][1].millitm;
                if (avg.millitm % 1000 != avg.millitm)
                        {
                        avg.time += (long)(avg.millitm / (short)1000);
                        avg.millitm %= (short)1000;
                        }
                }
        sprintf(tmp, "%s Tot. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));
        i = (int) (avg.time % (long)NUMTIMES);
        avg.time /= (long)NUMTIMES;
        avg.millitm += i * 1000;
        avg.millitm /= NUMTIMES;
        sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));

        for(i=0, avgTimeTaken=0; i<NUMTIMES; i++)
                avgTimeTaken+=timeTaken[i];
        avgTimeTaken = avgTimeTaken / NUMTIMES;
        sprintf(tmp, "%s Avg. Time Taken:%d\n", mapFileName, avgTimeTaken);
        avgFile.write(tmp, strlen(tmp));
        logFile.write(tmp, strlen(tmp));

        for(i=0, avgDist=0.0; i<NUMTIMES; i++)
```

```
                avgDist += distRobotTravelled[i];
            avgDist = avgDist / ((float)NUMTIMES);
            sprintf(tmp, "%s Avg. Dist Travelled:%f\n", mapFileName, avgDist);
            avgFile.write(tmp, strlen(tmp));
            logFile.write(tmp, strlen(tmp));

            sprintf(tmp, "%s ====================\n", mapFileName);
            avgFile.write(tmp, strlen(tmp));
//          sprintf(tmp, "%s ====================\n", mapFileName);
            logFile.write(tmp, strlen(tmp));
            }

// domain.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00

#define FROM_FRONT 0
#define FROM_BACK 1
#define FROM_LEFT 2
#define FROM_RIGHT 3
#define FROM_FRONTRIGHT 4
#define FROM_FRONTLEFT 5
#define FROM_BACKRIGHT 6
#define FROM_BACKLEFT 7
#define FROM_UP 8
#define FROM_DOWN 9
#define FROM_NOWHERE 10

#define NOCOST (float)0.0
#define NORMAL (float)1.0
#define NORMAL_DIAG (float)1.414214
#define BLOCKED (float)1000.0

typedef struct
        {
        float dist;
        int from;
        char type;
        float cost[9];
        } POINT;

class Domain
        {
        public:
                Domain(int, int, int, char*);
                ~Domain();
                void AdvanceTime(void);
        float CheckLine(int, int, int, int, int, int);
                void DrawDomain(void);
        int GetDomainLength(void) { return(domainLength); }
        int GetDomainTimeSlices(void) { return(domainTimeSlices); }
        int GetDomainWidth(void) { return(domainWidth); }
                POINT* GetPoint(int, int, int);
        float GetPointCost(int, int, int, int);
                char GetPointFrom(int, int, int);
                char GetPointType(int, int, int);
                int IsPointClear(int, int, int);
                int IsPointGoal(int, int, int);
                int IsPointNearObject(int, int, int);
                int IsPointObject(int, int, int);
                int IsPointStart(int, int, int);
        int IsPointVertex(int, int, int);
                int MoveRobot(int, int, int, int, int, int, float*);
                void SetPointFrom(int, int, int, int);
                void SetPointType(int, int, int, char);
        private:
                void AgeTimeSlices(void);
        int CalcRobotDir(int, int, int, int, int, int);
                int ClearAdjPointOK(int, int, int);
                void ClearMobileObject(int, int, int);
```

```
        void ClearVerticesInTimeSlice(int);
        void DrawTimeSlice(int);
        void InitTimeSlice(int);
        void MarkMobileObject(int, int, int);
        void MoveMobileObject(OBJECT_NODE*);
        void MoveMobileObjects(void);
        void SetAdjObjsInTimeSlice(int);
        void SetGoalFromFile(char*);
        void SetMobileObjsFromFile(char*);
        void SetPermObjsInTimeSlice(int);
        void SetStartFromFile(char*);
        void SetVerticesInTimeSlice(int);

        ObjectList objList;
        POINT** domainHead;
    int domainWidth;
    int domainLength;
    int domainTimeSlices;
    };

// domain.cpp
#include <iostream.h>
#include <fstream.h>
#include <alloc.h>          // for coreleft()
#include <stdio.h>
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <conio.h>
#include "object.hpp"
#include "domain.hpp"

Domain::Domain(int numTimeSlices, int width, int length, char*mapFileName)
    {
    int i;

    cout << "Contructing Domain class\n";
    domainTimeSlices = numTimeSlices;
    domainWidth = width;
    domainLength = length;

    cout << "FreeHeap:" << farcoreleft() << "\n";
    cout << "Amount needed for one timeslice:";
    cout << width*length*sizeof(POINT) << "\n";

    domainHead = (POINT**)farcalloc(numTimeSlices, sizeof(POINT**));
    if (domainHead == (POINT**)NULL)
        {
        cout << "Not enough memory to build model of world\n";
        domainHead = (POINT***)NULL;
        domainTimeSlices = domainWidth = domainLength = 0;
        return;
        }

    for(i=0; i<numTimeSlices; i++)
        {
        domainHead[i] = (POINT*)farcalloc(width*length, sizeof(POINT));
        cout << "FreeHeap after timeslice allocated:" << farcoreleft() << "\n";
        if (domainHead[i] == (POINT*)NULL)
            {
            if (i==0)
                cout << "Not enough memory to build model of world\n";
            else
                cout << "Only enough memory to build " << i << " of the ";
                cout << numTimeSlices << " timeslices in model of world\n";
            domainTimeSlices = i;
            break;
            }
        InitTimeSlice(i);
        SetPermObjsInTimeSlice(i);
        }
    SetStartFromFile(mapFileName);
    SetGoalFromFile(mapFileName);
    SetMobileObjsFromFile(mapFileName);
    for(i=0; i<domainTimeSlices; i++)
        {
        SetAdjObjsInTimeSlice(i);
        SetVerticesInTimeSlice(i);
        }
    for(i=0; i<domainTimeSlices-1; i++)
        AdvanceTime();
    cout << "FreeHeap after Domain allocated:" << farcoreleft() << "\n";
    }
```

```
Domain::~Domain()
        {
        POINT *tmp;
        int i;

        cout << "Destructing Domain class\n";
        if (domainHead == (POINT**)NULL)
                {
                cout << "Not freeing domain - DomainHead was NULL\n";
                return;
                }
        for(i=0; i<domainTimeSlices; i++)
                {
                tmp = domainHead[i];
                farfree(tmp);
                }
        farfree(domainHead);
        domainHead = (POINT**)NULL;
        cout << "FreeHeap after Domain deallocated:" << farcoreleft() << "\n";
//      getch();
        }


void Domain::AdvanceTime()
        {
        AgeTimeSlices();
//      DrawDomain();
        ClearVerticesInTimeSlice(domainTimeSlices-1);
        MoveMobileObjects();
//      DrawDomain();
        SetVerticesInTimeSlice(domainTimeSlices-1);
//      DrawDomain();
        }


//////////////////////////////////////
//
// Copy the contents of every timeslice
// into the previous timeslice
// [0] <= [1], [1] <= [2], etc.
// Leave the last timeslice unchanged.
// Another routine will decide the moves
// for all the mobile objects.
//
// A quick way to do this is moving ptrs
// to the timeslices and only copying the
// contents of the last timeslice over the
// contents of the first timeslice
//
//////////////////////////////////////
void Domain::AgeTimeSlices()
        {
        int i, j;
        POINT *tmp;

        tmp = domainHead[0];
        for(i=0, j=1; i<domainTimeSlices-1; i++, j++)
                domainHead[i] = domainHead[j];
        domainHead[domainTimeSlices-1] = tmp;
        _fmemcpy(domainHead[domainTimeSlices-1],
                                domainHead[domainTimeSlices-2],
                                sizeof(*domainHead[domainTimeSlices-1]));
        }


int Domain::CalcRobotDir(int st, int sw, int sl, int et, int ew, int el)
        {
        if ((!IsPointStart(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(clear);

        if ((ew - sw > 0)
        && (el - sl > 0))
                {
                sw++, sl++;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
                || (IsPointGoal(st, sw, sl)))
                        return(frontright);
                else
                        return(clear);
                }
```

```
if ((ew - sw > 0)
&& (el - sl == 0))
        {
        sw++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(right);
        else
                return(clear);
        }

if ((ew - sw > 0)
&& (el - sl < 0))
        {
        sw++, sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(backright);
        else
                return(clear);
        }

if ((ew - sw == 0)
&& (el - sl > 0))
        {
        sl++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(front);
        else
                return(clear);
        }

if ((ew - sw == 0)
&& (el - sl == 0)
&& (et - st > 0))
        {
        st++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(up);
        else
                return(clear);
        }

if ((ew - sw == 0)
&& (el - sl < 0))
        {
        sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(back);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl > 0))
        {
        sw--, sl++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(frontleft);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl == 0))
        {
        sw--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(left);
        else
                return(clear);
        }
```

```
        if ((ew - sw < 0)
        && (el - sl < 0))
                {
                sw--, sl--;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
                || (IsPointGoal(st, sw, sl)))
                        return(backleft);
                else
                        return(clear);
                }
        return(clear);
        }


float Domain::CheckLine(int st, int sw, int sl, int et, int ew, int el)
        {
//      static int displayCounter=1;
        int tmpt, tmpw, tmpl;
        float dist;

        for(dist=0.0, tmpt=et-st, tmpw=ew-sw, tmpl=el-sl;
                (tmpt != 0) || (tmpw != 0) || (tmpl != 0);
                tmpt=et-st, tmpw=ew-sw, tmpl=el-sl)
                {
                if (tmpt < 0)
                        {
                        return(-1.0);
                        }
                if ((tmpw > 0)
                && (tmpl > 0))
                        {
                        dist += GetPointCost(st, sw++, sl++, frontright);
                        if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                                return(-1.0);
          continue;
                        }
       if ((tmpw > 0)
                && (tmpl == 0))
                        {
          dist += GetPointCost(st, sw++, sl, right);
                if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                        return(-1.0);
          continue;
                        }
                if ((tmpw > 0)
                && (tmpl < 0))
                        {
          dist += GetPointCost(st, sw++, sl--, backright);
                        if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                        return(-1.0);
          continue;
          }
                if ((tmpw < 0)
                && (tmpl > 0))
                        {
          dist += GetPointCost(st, sw--, sl++, frontleft);
                if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                        return(-1.0);
          continue;
          }
                if ((tmpw < 0)
                && (tmpl == 0))
                        {
          dist += GetPointCost(st, sw--, sl, left);
                if ((!IsPointClear(st, sw, sl))
                        && ((st != et) || (sw != ew) || (sl != el)))
                        return(-1.0);
          continue;
          }
                if ((tmpw < 0)
                && (tmpl < 0))
                        {
          dist += GetPointCost(st, sw--, sl--, backleft);
                        if ((!IsPointClear(st, sw, sl))
                        && ((st != et) || (sw != ew) || (sl != el)))
                                return(-1.0);
          continue;
```

```
                |
           if ((tmpw == 0)
           && (tmpl > 0))
                |
        dist += GetPointCost(st, sw, sl++, front);
           if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
        continue;
        }
           if ((tmpw == 0)
           && (tmpl < 0))
                {
        dist += GetPointCost(st, sw, sl--, back);
           if ((!IsPointClear(st, sw, sl))
           && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
        continue;
        }
      if ((tmpw == 0)
           && (tmpl == 0)
           && (tmpt >0))
                {
        dist += GetPointCost(st++, sw, sl, up);
           if ((!IsPointClear(st, sw, sl))
           && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
        continue;
        }
           }
//     cout << "CheckLine:" << displayCounter++ << ":returned" << dist << "\n";
       return(dist);
       }


int Domain::ClearAdjPointOK(int t, int w, int l)
       {
       char type;
       int a,b;

       for(a=w-1;a<=w+1;a++)
           {
           if ((a < 0) || (a >= domainWidth))
                continue;
           for(b=l-1;b<=l+1;b++)
                {
                if ((b < 0) || (b >= domainLength))
                     continue;
                if ((a == w) && (b == l))
                     continue;

         type = GetPointType(t, a, b);
                if ((type == OBJECT)
                ||  (type == MOBILE_OBJECT))
                     return(FALSE);
                }

           }
       return(TRUE);
       }


void Domain::ClearMobileObject(int t, int w, int l)
   {
       int i, j;

       if (GetPointType(t, w, l) == MOBILE_OBJECT)
           {
           if (ClearAdjPointOK(t, w, l))
                {
                SetPointType(t, w, l, CLEAR);
                SetPointFrom(t, w, l, FROM_NOWHERE);
                }
           else
                {
                SetPointType(t, w, l, ADJ_TO_OBJECT);
                SetPointFrom(t, w, l, FROM_NOWHERE);
                }
           }
       for(i=w-1, j=l-1; j<l+2;)
           {
      if ((GetPointType(t, i, j) == ADJ_TO_OBJECT)
           && (ClearAdjPointOK(t, i, j)))
           {
           SetPointType(t, i, j, CLEAR);
```

```
                    SetPointFrom(t, i, j, FROM_NOWHERE);
                    }
            if (i == w+1)
                    {
                    i = w-1;
                    j++;
                    }
            else
                    i++;
            }
        }


void Domain::ClearVerticesInTimeSlice(int t)
        {
        int w, l, pointType;

        for(w=0; w<domainWidth; w++)
                {
                for(l=0; l<domainLength; l++)
                        {
                        pointType = GetPointType(t, w, l);
                        if (pointType == VERTEX)
                                {
                                SetPointFrom(t, w, l, FROM_NOWHERE);
                                SetPointType(t, w, l, CLEAR);
                                continue;
                                }
                        if (pointType == MOBILE_OBJECT)
                                {
            ClearMobileObject(t, w, l);
                                }
                        }
                }
        }


void Domain::DrawDomain()
        {
        int i;

        for (i=0; i< domainTimeSlices; i++)
                {
                DrawTimeSlice(i);
//              getch();
                }
        }


void Domain::DrawTimeSlice(int timeSlice)
        {
        char type;
        int a,b,y;

        clrscr();
        gotoxy(1,1);
        cout << "time=t+" << timeSlice;

        for(b=0, y=2;b<domainLength; b++, y++)
                {
                gotoxy(1, y);
                for(a=0; a<domainWidth; a++)
                        {
                        type = GetPointType(timeSlice, a, b);
                        switch(type)
                                {
                                case GOAL:
                                case START:
                                case ADJ_TO_OBJECT:
                                case OBJECT:
                                case MOBILE_OBJECT:
                                case VERTEX:
                                        cout << type;
                                        break;
                                case CLEAR:
                                        switch(GetPointFrom(timeSlice, a, b))
                                                {
                                                case FROM_RIGHT:
                                                        cout << "R";
                                                        break;
                                                case FROM_LEFT:
                                                        cout << "L";
                                                        break;
                                                case FROM_UP:
```

```
                                                cout << "U";
                                                break;
                                        case FROM_DOWN:
                                                cout << "D";
                                                break;
                                        case FROM_FRONT:
                                                cout << "F";
                                                break;
                                        case FROM_BACK:
                                                cout << "B";
                                                break;
                                        case FROM_BACKLEFT:
                                                cout << "T";
                                                break;
                                        case FROM_BACKRIGHT:
                                                cout << "U";
                                                break;
                                        case FROM_FRONTLEFT:
                                                cout << "V";
                                                break;
                                        case FROM_FRONTRIGHT:
                                                cout << "W";
                                                break;
                                        case FROM_NOWHERE:
                                        default:
                                                cout << ".";
                                                break;
                                        }
                                break;
                        default:
                                cout << "?";
                                break;
                        }
                }
        cout << "\n";
        }
    }


POINT* Domain::GetPoint(int timeSlice, int width, int length)
    {
  POINT *tmp, *tmp2;

  tmp = domainHead[timeSlice];
    tmp2 = tmp + (domainWidth*width) + length;
  return(tmp2);
  }


float Domain::GetPointCost(int timeSlice, int width, int length, int dir)
  {
  POINT *tmp;

  tmp = GetPoint(timeSlice, width, length);
    return(tmp->cost[dir]);
    }

char Domain::GetPointFrom(int timeSlice, int width, int length)
  {
  POINT *tmp;

  tmp = GetPoint(timeSlice, width, length);
    return(tmp->from);
  }


char Domain::GetPointType(int timeSlice, int width, int length)
    {
  POINT *tmp;

    tmp = GetPoint(timeSlice, width, length);
    return(tmp->type);
    }


void Domain::InitTimeSlice(int timeSlice)
    {
    POINT*tmp;
  int a,b;

    for(a=0;a<domainWidth;a++)
        {
        for(b=0;b<domainLength;b++)
            {
```

```
tmp = GetPoint(timeSlice, a, b);
          tmp->dist=0.0;
          tmp->from=FROM_NOWHERE;
          tmp->type=CLEAR;

/*        if (timeSlice == domainTimeSlices-1)
                tmp->cost[FROM_UP] = BLOCKED;
          else*/
                tmp->cost[FROM_UP] = NORMAL;

          if ((a == 0) && (b == 0))
                {
                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                tmp->cost[FROM_BACK] = BLOCKED;
                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                tmp->cost[FROM_LEFT] = BLOCKED;
                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
          if ((a == 0) && (b < domainLength-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                tmp->cost[FROM_LEFT] = BLOCKED;
                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
          if ((a == 0) && (b==domainLength-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                tmp->cost[FROM_LEFT] = BLOCKED;
                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                tmp->cost[FROM_FRONT] = BLOCKED;
                continue;
                }
          if ((a < domainWidth-1) && (b == 0))
                {
                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                tmp->cost[FROM_BACK] = BLOCKED;
                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                tmp->cost[FROM_LEFT] = NORMAL;
                tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
          if ((a < domainLength-1) && (b < domainWidth-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_LEFT] = NORMAL;
                tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
          if ((a < domainWidth-1) && (b == domainLength-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_LEFT] = NORMAL;
                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                tmp->cost[FROM_FRONT] = BLOCKED;
                continue;
                }
          if ((a == domainWidth-1) && (b == 0))
                {
```

```
                                        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                                        tmp->cost[FROM_RIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACK] = BLOCKED;
                                        tmp->cost[FROM_BACKLEFT] = BLOCKED;
                                        tmp->cost[FROM_LEFT] = NORMAL;
                                        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                                        tmp->cost[FROM_FRONT] = NORMAL;
                                        continue;
                                        }
                                if ((a == domainWidth-1) && (b < domainWidth-1))
                                        {
                                        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                                        tmp->cost[FROM_RIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACK] = NORMAL;
                                        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                                        tmp->cost[FROM_LEFT] = NORMAL;
                                        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                                        tmp->cost[FROM_FRONT] = NORMAL;
                                        continue;
                                        }
                                if ((a == domainLength-1) && (b == domainWidth-1))
                                        {
                                        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                                        tmp->cost[FROM_RIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                        tmp->cost[FROM_BACK] = NORMAL;
                                        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                                        tmp->cost[FROM_LEFT] = NORMAL;
                                        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                                        tmp->cost[FROM_FRONT] = BLOCKED;
                                        continue;
                                        }
                                }
                        }
                }


int Domain::IsPointClear(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case CLEAR:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointGoal(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case GOAL:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }



int Domain::IsPointNearObject(int t, int w, int l)
        {
        if ((w < 0)
```

```
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
                case OBJECT:
                        return(OBJECT);
                case MOBILE_OBJECT:
                        return(MOBILE_OBJECT);
                case ADJ_TO_OBJECT:
                        return(ADJ_TO_OBJECT);
                default:
                        return(FALSE);
                }
        }

int Domain::IsPointObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
                case OBJECT:
                        return(OBJECT);
                case MOBILE_OBJECT:
                        return(MOBILE_OBJECT);
                case ADJ_TO_OBJECT:
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointStart(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case START:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointVertex(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case VERTEX:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


void Domain::MarkMobileObject(int t, int w, int l)
        {
        int i, j;
```

```
	if ((GetPointType(t, w, l) == CLEAR)
	|| (GetPointType(t, w, l) == ADJ_TO_OBJECT))
		{
		SetPointType(t, w, l, MOBILE_OBJECT);
		SetPointFrom(t, w, l, FROM_NOWHERE);
		}
	for(i=w-1, j=l-1; j<l+2;)
		{
		if (GetPointType(t, i, j) == CLEAR)
			{
			SetPointType(t, i, j, ADJ_TO_OBJECT);
			SetPointFrom(t, i, j, FROM_NOWHERE);
			}
		if (i == w+1)
			{
			i = w-1;
			j++;
			}
		else
			i++;
		}
	}


void Domain::MoveMobileObject(OBJECT_NODE*object)
	{
	int t=domainTimeSlices-1, w=(*object).w, l=(*object).l;

	ClearMobileObject(t, w, l);
	switch((*object).direction)
		{
		case front:
			if ((IsPointObject(t, w, l+1))
			|| (IsPointStart(t, w, l+1))
			|| (IsPointGoal(t, w, l+1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).l++;
			break;
		case frontleft:
			if ((IsPointObject(t, w-1, l+1))
			|| (IsPointStart(t, w-1, l+1))
			|| (IsPointGoal(t, w-1, l+1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).l++, (*object).w--;
			break;
		case left:
			if ((IsPointObject(t, w-1, l))
			|| (IsPointStart(t, w-1, l))
			|| (IsPointGoal(t, w-1, l)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w--;
			break;
		case backleft:
			if ((IsPointObject(t, w-1, l-1))
			|| (IsPointStart(t, w-1, l-1))
			|| (IsPointGoal(t, w-1, l-1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w--, (*object).l--;
			break;
		case back:
			if ((IsPointObject(t, w, l-1))
			|| (IsPointStart(t, w, l-1))
			|| (IsPointGoal(t, w, l-1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).l--;
			break;
		case backright:
			if ((IsPointObject(t, w+1, l-1))
			|| (IsPointStart(t, w+1, l-1))
			|| (IsPointGoal(t, w+1, l-1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w++, (*object).l--;
			break;
		case right:
			if ((IsPointObject(t, w+1, l))
			|| (IsPointStart(t, w+1, l))
			|| (IsPointGoal(t, w+1, l)))
				(*object).direction = rand() % NUM_DIRS;
```

```
                        else
                                (*object).w++;
                        break;
                case frontright:
                        if ((IsPointObject(t, w+1, l+1))
                        || (IsPointStart(t, w+1, l+1))
                        || (IsPointGoal(t, w+1, l+1)))
                                (*object).direction = rand() % NUM_DIRS;
                        else
                                (*object).w++, (*object).l++;
                        break;
                default:
                        cout << "*** unknown direction for mobile object ignored ***\n";
                        break;
                }
        MarkMobileObject(t, object->w, object->l);
        }


void Domain::MoveMobileObjects()
        {
        OBJECT_NODE*cur, *orig;

        orig = cur = objList.GetNextObject();
        if (cur == (OBJECT_NODE*)NULL)
       return;

        do
                {
                if ((*cur).velocity > 0)
                        MoveMobileObject(cur);

       cur = objList.GetNextObject();
                } while (cur != orig);
        }


int Domain::MoveRobot(int st, int sw, int sl, int et, int ew, int el, float*dist)
        {
        int dir;

        *dist = 0.0;
        if (!IsPointStart(st, sw, sl))
                {
                *dist = 0.0;
                return(FALSE);
                }

        dir = CalcRobotDir(st, sw, sl, et, ew, el);
        SetPointType(st, sw, sl, CLEAR);
        SetPointFrom(st, sw, sl, FROM_NOWHERE);

        switch(dir)
                {
                case frontright:
                        *dist = GetPointCost(st, sw, sl, frontright);
                        st++, sw++, sl++;
                        if ((!IsPointClear(st, sw, sl))
                        && (!IsPointVertex(st, sw, sl))
                        && (!IsPointGoal(st, sw, sl)))
                                {
                                st--, sw--, sl--;
                                *dist = 0.0;
                                }
                        break;
                case right:
                        *dist = GetPointCost(st, sw, sl, right);
                        st++, sw++;
                        if ((!IsPointClear(st, sw, sl))
                        && (!IsPointVertex(st, sw, sl))
                        && (!IsPointGoal(st, sw, sl)))
                                {
                                st--, sw--;
                                *dist = 0.0;
                                }
                        break;
                case backright:
                        *dist = GetPointCost(st, sw, sl, backright);
                        st++, sw++, sl--;
                        if ((!IsPointClear(st, sw, sl))
                        && (!IsPointVertex(st, sw, sl))
                        && (!IsPointGoal(st, sw, sl)))
                                {
```

```
				st--, sw--, sl++;
				*dist = 0.0;
				}
			break;
		case frontleft:
			*dist = GetPointCost(st, sw, sl, frontleft);
			st++, sw--, sl++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw++, sl--;
				*dist = 0.0;
				}
			break;
		case left:
			*dist = GetPointCost(st, sw, sl, left);
			st++, sw--;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw++;
				*dist = 0.0;
				}
			break;
		case backleft:
			*dist = GetPointCost(st, sw, sl, backleft);
			st++, sw--, sl--;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw++, sl++;
				*dist = 0.0;
				}
			break;
		case front:
			*dist = GetPointCost(st, sw, sl, front);
			st++, sl++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sl--;
				*dist = 0.0;
				}
			break;
		case back:
			*dist = GetPointCost(st, sw, sl, back);
			st++, sl--;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sl++;
				*dist = 0.0;
				}
			break;
		case up:
			*dist = GetPointCost(st, sw, sl, up);
			st++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--;
				*dist = 0.0;
				}
			break;
		default:	// no movement - start IS goal
			*dist = 0.0;
			break;
		}
	if (IsPointGoal(st, sw, sl))
		return(TRUE);
	else
		{
		SetPointType(st, sw, sl, START);
		SetPointFrom(st, sw, sl, FROM_NOWHERE);
		return(FALSE);
		}
}
```

```
void Domain::SetAdjObjsInTimeSlice(int timeSlice)
	{
	int a,b;
	int w,l;

	for(w=0; w<domainWidth; w++)
		{
		for(l=0; l<domainLength; l++)
			{
			if (!IsPointObject(timeSlice, w, l))
				continue;

			for(a=w-1; a<=w+1; a++)
				{
				if ((a < 0) || (a >= domainWidth))
					continue;
				for(b=l-1; b<=l+1; b++)
					{
					if ((b < 0) || (b >= domainLength))
						continue;

					if (GetPointType(timeSlice, a, b) == CLEAR)
						{
						SetPointFrom(timeSlice, a, b, FROM_NOWHERE);
						SetPointType(timeSlice, a, b, ADJ_TO_OBJECT);
						}

				}
			}
		}
	}


void Domain::SetGoalFromFile(char*fileName)
	{
	int i, w, l;
	char recType;
	char tmp[256];
	fstream dataFile;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Goal added.\n";

	for(;;)
		{
		dataFile. getline(tmp, sizeof(tmp));
		if (strlen(tmp) == 0)
			break;
		if (tmp[0] == GOAL)
			{
//			cout << "The GOAL line is: " << tmp << "\n";
			if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
				{
		for(i=0; i<domainTimeSlices; i++)
		  {
				SetPointFrom(i, w, l, FROM_NOWHERE);
				SetPointType(i, w, l, GOAL);
		  }
				break;
				}
			else
				cout << "Improperly formatted line ignored\n";
			}
		}
	dataFile.close();
	}


void Domain::SetMobileObjsFromFile(char*fileName)
	{
	int w, l;
	char recType;
	char tmp[256];
	fstream dataFile;
	OBJECT_NODE* newObj;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Mobile objs added.\n";

	for(;;)
		{
```

```
		int i;

		dataFile.getline(tmp, sizeof(tmp));
		i = strlen(tmp);
		if (i <= 0)
			break;
		cout << "Length of input line is: " << i << "\n";
		if ((tmp[0] != GOAL)
		&& (tmp[0] != START))
			{
//			cout << "OBJECT line is: " << tmp << "\n";
			if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
				{
				SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
				SetPointType(domainTimeSlices-1, w, l, MOBILE_OBJECT);
				newObj = objList.BuildNewObject(w, l);
				if (newObj)
					{
					objList.InsertNewObject(newObj);
//					cout << "Added obj to obj list\n";

				}
			else
				cout << "Improperly formatted line ignored\n";
			}
		}
	dataFile.close();
	}


void Domain::SetPermObjsInTimeSlice(int timeSlice)
	{
	int w, l;

	for(w=0; w<domainWidth-5; w++)
		{
		l=2;
		switch(w)
			{
			case 3:
			case 4:
			case 5:
				break;
			default:
				SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
				SetPointType(timeSlice, w, l, OBJECT);
//				SetAdjObjsInTimeSlice(timeSlice, w, l);
				break;
			}
		}
	for(w=0; w<domainWidth-10; w++)
		{
		l=7;
		SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
		SetPointType(timeSlice, w, l, OBJECT);
//		SetAdjObjsInTimeSlice(timeSlice, w, l);
		}
	for(l=0, w=domainWidth-10; l<domainLength-2; l++)
		{
		switch(l)
			{
			case 3:
			case 4:
			case 5:
			case 6:
			case 14:
			case 15:
			case 16:
				break;
			default:
				SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
				SetPointType(timeSlice, w, l, OBJECT);
//				SetAdjObjsInTimeSlice(timeSlice, w, l);
				break;
			}
		}
	for(l=2, w=domainWidth-5; l<domainLength-7; l++)
		{
		switch(l)
			{
			case 9:
			case 10:
			case 11:
				break;
```

```
                        default:
                               SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                               SetPointType(timeSlice, w, l, OBJECT);
//                             SetAdjObjsInTimeSlice(timeSlice, w, l);
                               break;
                        }
                }
        for(w=domainWidth-5, l=domainLength-7; w<domainWidth; w++)
                {
                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                SetPointType(timeSlice, w, l, OBJECT);
//              SetAdjObjsInTimeSlice(timeSlice, w, l);
                }
        for(w=domainWidth-10, l=domainLength-2; w<domainWidth; w++)
                {
                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                SetPointType(timeSlice, w, l, OBJECT);
//              SetAdjObjsInTimeSlice(timeSlice, w, l);
                }
        }


void Domain::SetPointFrom(int timeSlice, int width, int length, int from)
        {
        POINT *tmp;

        if ((width < 0)
        || (length < 0)
        || (timeSlice < 0)
        || (width >= domainWidth)
        || (length >= domainLength)
        || (timeSlice >= domainTimeSlices))
                return;
        tmp = GetPoint(timeSlice, width, length);
        tmp->from = from;
        }

void Domain::SetPointType(int timeSlice, int width, int length, char type)
        {
        char oldType;
        POINT *tmp;

        if ((width < 0)
        || (length < 0)
        || (timeSlice < 0)
        || (width >= domainWidth)
        || (length >= domainLength)
        || (timeSlice >= domainTimeSlices))
                return;
        tmp = GetPoint(timeSlice, width, length);
        oldType = tmp->type;
        tmp->type = type;
        if (oldType != OBJECT)
                {
//              SetAdjObjsInTimeSlice(timeSlice, width, length);
                }
        }


void Domain::SetStartFromFile(char*fileName)
        {
        int w, l;
        char recType;
        char tmp[256];
        fstream dataFile;

        dataFile.open(fileName, ios::in);
        if (!dataFile)
                cout << "Unable to open " << fileName << ". No Start added.\n";

        for(;;)
                {
                dataFile. getline(tmp, sizeof(tmp));
                if (strlen(tmp) == 0)
                        break;
                if (tmp[0] == START)
                        {
//                      cout << "The START line is: " << tmp << "\n";
                        if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                                {
                                SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
                                SetPointType(domainTimeSlices-1, w, l, START);
                                break;
                                }
```

```
				else
					cout << "Improperly formatted line ignored\n";
				}
			}
		dataFile.close();
		}


void Domain::SetVerticesInTimeSlice(int t)
		{
		int w, l;
		int diff_counter, corner_counter;
		int side_bits, corner_bits;

		for(w=0; w<domainWidth; w++)
			{
			for(l=0; l<domainLength; l++)
				{
				if (!IsPointClear(t, w, l))
					continue;

				corner_bits = BITMASK_CLEAR;
				corner_counter = 0;
				if (IsPointNearObject(t, w-1, l-1))
					{
					corner_bits |= BITMASK_LEFT;
					corner_bits |= BITMASK_TOP;
					corner_counter++;
					}
				if (IsPointNearObject(t, w-1, l+1))
					{
					corner_bits |= BITMASK_LEFT;
					corner_bits |= BITMASK_BOTTOM;
					corner_counter++;
					}
				if (IsPointNearObject(t, w+1, l+1))
					{
					corner_bits |= BITMASK_RIGHT;
					corner_bits |= BITMASK_BOTTOM;
					corner_counter++;
					}
				if (IsPointNearObject(t, w+1, l-1))
					{
					corner_bits |= BITMASK_RIGHT;
					corner_bits |= BITMASK_TOP;
					corner_counter++;
					}
				if (corner_bits != BITMASK_CLEAR)
					{
					side_bits = BITMASK_CLEAR;
					if (IsPointNearObject(t, w-1, l))
						side_bits |= BITMASK_LEFT;
					if (IsPointNearObject(t, w, l-1))
						side_bits |= BITMASK_TOP;
					if (IsPointNearObject(t, w+1, l))
						side_bits |= BITMASK_RIGHT;
					if (IsPointNearObject(t, w, l+1))
						side_bits |= BITMASK_BOTTOM;

					for(diff_counter=0;
						(side_bits != BITMASK_CLEAR)
						|| (corner_bits != BITMASK_CLEAR);
						side_bits >>= 1, corner_bits >>= 1)
						{
						if ((corner_bits & (int)0x01)
						&& ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
							{
							diff_counter++;
							continue;
							}
						if ((side_bits & (int)0x01)
						&& ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
							{
							diff_counter++;
							continue;
							}
						}
					if ((diff_counter > 2)
					|| ((diff_counter == 2) && (corner_counter == 1)))
						{
						SetPointFrom(t, w, l, FROM_NOWHERE);
						SetPointType(t, w, l, VERTEX);
						}
					}
```

```
                    }
                }
            }

// edge.hpp
#define DEBUG_FILENAME "DEBUG.LOG"

typedef struct van
    {
        int t,w,l;
        float dist;
        struct van *prev, *next;
        } EDGE_NODE;

class EdgeList
        {
        public:
                EdgeList();
                ~EdgeList();
        EDGE_NODE* BuildNewEdge(int, int, int, float);
                void DelAllEdges(void);
                void DelEdge(EDGE_NODE*);
                void DelEdgeToVertex(int, int, int);
        EDGE_NODE* GetFirstEdge(void);
        EDGE_NODE* GetNextEdge(EDGE_NODE*);
        void InsertNewEdge(EDGE_NODE*);
                void ListAllEdges(int);
    private:
        EDGE_NODE *edgeHead;
        };

// edge.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include "edge.hpp"


EdgeList::EdgeList()
        {
        edgeHead = (EDGE_NODE*)NULL;
//    cout << "EdgeList Constructor\n";
        }


EdgeList::~EdgeList()
        {
        while (edgeHead != (EDGE_NODE*)NULL)
                DelEdge(edgeHead);
//      cout << "EdgeList Destructor\n";
        }


EDGE_NODE* EdgeList::BuildNewEdge(int t, int w, int l, float dist)
        {
        EDGE_NODE *a;

//      if (otherVertex != (void*)NULL)
//              {
                a = (EDGE_NODE*)farcalloc(1, sizeof(EDGE_NODE));
                if (a == (EDGE_NODE*)NULL)
                        {
                        cout << "Out of Memory in BuildNewEdge()\n";
          getch();
                        exit(0);
                        }
                (*a).t = t;
                (*a).w = w;
                (*a).l = l;
                (*a).dist = dist;
                (*a).prev = (*a).next = (EDGE_NODE*)NULL;
                return(a);
//              }
//      return((EDGE_NODE*)NULL);
        }


void EdgeList::DelAllEdges()
        {
        EDGE_NODE*a;
```

```
        for(a = GetFirstEdge();
                a != (EDGE_NODE*)NULL;
                a = GetFirstEdge())
                {
                DelEdge(a);
                }
        }

void EdgeList::DelEdge(EDGE_NODE*a)
        {
        EDGE_NODE*tmp;

//      cout << "edge 0\n";
        if (a != (EDGE_NODE*)NULL)
                {
                if ((a->prev == (EDGE_NODE*)NULL)  // del last remaining edge
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 1\n";
                        free(a);
                        edgeHead = (EDGE_NODE*)NULL;
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge in middle
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 2\n";
                        tmp = a->prev;
                        tmp->next = a->next;
                        tmp = a->next;
                        tmp->prev = a->prev;
                        free(a);
                        return;
                        }
                if ((a->prev == (EDGE_NODE*)NULL)  // del edge at sol
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 3\n";
                        tmp = a->next;
                        tmp->prev = (EDGE_NODE*)NULL;
                        edgeHead = tmp;
                        free(a);
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge at eol
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 4\n";
                        tmp = a->prev;
                        tmp->next = (EDGE_NODE*)NULL;
                        free(a);
                        return;
                        }
                }
        }

void EdgeList::DelEdgeToVertex(int t, int w, int l)
        {
        EDGE_NODE*tmp;

        for(tmp = GetFirstEdge(); tmp != (EDGE_NODE*)NULL; tmp = GetNextEdge(tmp))
                {
                if ((tmp->t == t)
        && (tmp->w == w)
        && (tmp->l == l))
                        {
                        DelEdge(tmp);
                        break;
                        }
                }
        }

EDGE_NODE* EdgeList::GetFirstEdge()
        {
    return(edgeHead);
    }

EDGE_NODE* EdgeList::GetNextEdge(EDGE_NODE*cur)
    {
    return(cur->next);
```

```
	}

void EdgeList::InsertNewEdge(EDGE_NODE *edge)
	{
	EDGE_NODE*cur;

	if (edge == (EDGE_NODE*)NULL)
		return;

	if (edgeHead == (EDGE_NODE*)NULL)
		{
		edgeHead = edge;
		edge->prev = edge->next = (EDGE_NODE*)NULL;
//		cout << "Inserted edge into empty list ";
//		cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//		cout << "Dist = " << edge->dist << "\n";
		return;
		}

	for(cur=edgeHead; cur != (EDGE_NODE*)NULL; cur=(*cur).next)
		{
		if ( (edge->t > cur->t)
		|| ((edge->t == cur->t)
			&& (edge->w > cur->w))
		|| ((edge->t == cur->t)
			&& (edge->w == cur->w)
			&& (edge->l > cur->l)) )
			{				// insert after cur
			if (cur->next == (EDGE_NODE*)NULL)
				{
				edge->next = (EDGE_NODE*)NULL;// no more so append to eol
				edge->prev = cur;
				cur->next = edge;
				break;
				}
			else
				continue;			// try next one
			}
		if ( (edge->t < cur->t)
		|| ((edge->t == cur->t)
			&& (edge->w < cur->w))
/*		|| ((edge->t < cur->t)*/
		|| ((edge->t == cur->t)
			&& (edge->w == cur->w)
			&& (edge->l < cur->l)) )
			{				// insert before cur
			if (cur->prev == (EDGE_NODE*)NULL)
				{
				edge->prev = (EDGE_NODE*)NULL;	// at start of list
				edge->next = cur;
				cur->prev = edge;
				edgeHead = edge;
				break;
				}
			else
				{
				edge->prev = cur->prev;		// in middle/end of list
				edge->next = cur;
				cur->prev = edge;
				cur = edge->prev;
				cur->next = edge;
				break;
				}
			}
		if ( (edge->t == cur->t)
		&& (edge->w == cur->w)
		&& (edge->l == cur->l) )
			{
			/* already here - replace it ! */
/*			cout << "Already here - replacing values and disposing of new edge ";
			cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
			cout << "Dist = " << edge->dist << "\n";
			cur->w = edge->w;
			cur->l = edge->l;
			cur->dist = edge->dist;*/
//			cout << " Already here - ignoring...\n";
			cout << "!";
			free(edge);
			break;
			}
		}
//	cout << "Inserted edge into list ";
//	cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
```

```
//      cout << "Dist = " << edge->dist << "\n";
        }


void EdgeList::ListAllEdges(int debugFlag)
        {
        char tmp[256];
        fstream debugFile;
        EDGE_NODE *edge;

        strcpy(tmp, "List of all edges in list\n");
        cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        for(edge = edgeHead; edge != (EDGE_NODE*)NULL; edge = (*edge).next)
                {
                sprintf(tmp, "Edge to (%d,%d,%d). Dist is %f\n",
                                                (*edge).t, (*edge).w, (*edge).l, (*edge).dist);
                cout << tmp;
                if (debugFlag)
                        {
                        debugFile.open(DEBUG_FILENAME, ios::app);
                        debugFile.write(tmp, strlen(tmp));
                        debugFile.close();
                        }
                }
        strcpy(tmp, "-----------------\n");
        cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        }

// object.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define NUM_DIRS 7 /* this is the 8 horizontal directions; (0->7) */
enum directions { front, back, left, right, frontright, frontleft, backright, backleft, up, down, clear};


typedef struct o
        {
        int direction, velocity;
        int w, l;
        struct o *prev, *next;
        } OBJECT_NODE;


class ObjectList
        {
        public:
                ObjectList();
                ~ObjectList();
                OBJECT_NODE* BuildNewObject(int, int);
        void DelAllObjects(void);
        void DelObject(OBJECT_NODE*);
        OBJECT_NODE* GetNextObject(void);
                void InsertNewObject(OBJECT_NODE*);
        void ListAllObjects(void);
        private:
                OBJECT_NODE *objectHead;
        };

// object.cpp
#include <iostream.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
```

```
#include <string.h>         // for strcpy()
#include <conio.h>
// #include "edge.hpp"
#include "object.hpp"

ObjectList::ObjectList()
        {
        objectHead = (OBJECT_NODE*)NULL;
        cout << "ObjectList Constructor\n";
        }



ObjectList::~ObjectList()
        {
        cout << "ObjectList Destructor\n";
        DelAllObjects();
        }



OBJECT_NODE* ObjectList::BuildNewObject(int w, int l)
        {
        OBJECT_NODE *newPtr;

        newPtr = (OBJECT_NODE*)farcalloc(1, sizeof(OBJECT_NODE));
        if (newPtr == (OBJECT_NODE*)NULL)
                {
                cout << "Out of memory in BuildNewObject()\n";
       getch();
                return(NULL);
                }
        (*newPtr).direction = rand() % NUM_DIRS;
        (*newPtr).velocity = 1;
        (*newPtr).prev = (*newPtr).next = (OBJECT_NODE*)NULL;
        (*newPtr).w = w;
        (*newPtr).l = l;
        return(newPtr);
        }



void ObjectList::DelAllObjects()
        {
        OBJECT_NODE *tmp;

        while (objectHead != (OBJECT_NODE*)NULL)
                {
                tmp = objectHead;
                objectHead = (*objectHead).next;
                DelObject(tmp);
                }
        }



void ObjectList::DelObject(OBJECT_NODE *todie)
   {
        OBJECT_NODE *cur;

        if (((*todie).prev != (OBJECT_NODE*)NULL)
        && ((*todie).next != (OBJECT_NODE*)NULL))
                {
//              cout << "Deleted Object (" << todie->w << "," << todie->l << ")\n";
                cur=(*todie).prev;
                if (cur == todie)
                        {
                        objectHead = (OBJECT_NODE*)NULL;
//                      cout << "Object List Empty\n";
                        free(todie);
                        return;
                        }
                else
                        {
                        (*cur).next=(*todie).next;
                        cur = (*todie).next;
                        (*cur).prev = (*todie).prev;
                        if (objectHead == todie)
                                objectHead = (*todie).next;
       free(todie);
         return;
                        }
                }
       cout << "***Did NOT delete rotton Object (" << todie->w << "," << todie->l << ")\n";
```

```
        }

OBJECT_NODE* ObjectList::GetNextObject()
    {
    OBJECT_NODE*tmp;

        if (objectHead == (OBJECT_NODE*)NULL)
                return((OBJECT_NODE*)NULL);

        tmp = objectHead;
    objectHead = objectHead->next;
    return(tmp);
    }


void ObjectList::InsertNewObject(OBJECT_NODE*newPtr)
        {
        OBJECT_NODE *cur;

        if (newPtr == (OBJECT_NODE*)NULL)
                return;

        if (objectHead == (OBJECT_NODE*)NULL)
                {
                objectHead = newPtr;
                (*newPtr).prev = (*newPtr).next = newPtr;
                }
        else
                {
                /* insert before first node */
                cur = objectHead;
                (*newPtr).next = cur;
                (*newPtr).prev = (*cur).prev;
                (*cur).prev = newPtr;

                cur = (*newPtr).prev;
                (*cur).next = newPtr;
                objectHead = newPtr;
        }
//      cout << "Inserted object (" << newPtr->w << "," << newPtr->l << ")\n";
        }


void ObjectList::ListAllObjects()
       {
   OBJECT_NODE*tmp;

   if (objectHead == (OBJECT_NODE*)NULL)
       {
       cout << "ObjectList is empty\n";
       return;
       }
   for(tmp=objectHead;;tmp = tmp->next)
       {
        cout << "Object:(" << tmp->w << "," << tmp->l << ")\n";
       tmp = tmp->next;
       if (tmp == objectHead)
           {
           cout << "ObjectList ended\n";
           break;
           }
       }
        }

// vertex.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00
```

```
#define DEBUG_FILENAME "DEBUG.LOG"

typedef struct vn
	{
	int t,w,l;
	char nodeType;
	struct vn *searchPrev, *searchNext;
	float searchDist;
	int searchMarker;
	EdgeList *edgeList;
	struct vn *pathFrom, *pathTo;
	struct vn *prev, *next;
	} VERTEX_NODE;


class VertexList
	{
	public:
		VertexList();
		~VertexList();
		void AddToSearchList(VERTEX_NODE*);
	VERTEX_NODE* BuildNewVertex(int,int,int,char);
		int CalcRobotDir(VERTEX_NODE*);
		void DelAllVertices(void);
	void DelVertex(VERTEX_NODE*);
	int FindPath(void);
	VERTEX_NODE* FindVertex(int, int, int);
		VERTEX_NODE* GetFirstVertex(void);
		VERTEX_NODE* GetGoalVertex(void);
		VERTEX_NODE* GetNextVertex(VERTEX_NODE*);
		int GetSearchMarker(void);
	VERTEX_NODE* GetStartVertex(void);
	void InsertAllVertices(void);
		void InsertNewVertex(VERTEX_NODE*);
		void ListAllVertices(int);
	void ListSearchList(int);
//		void MoveRobot(int, int, int, int);
		void MarkPath(VERTEX_NODE*);
		void RemoveFromSearchList(VERTEX_NODE*);
	void RemoveVertex(VERTEX_NODE*);
		void TrimSearchList(void);
	private:
		VERTEX_NODE* vertexHead;
	VERTEX_NODE* searchHead;
	int searchMarker;
	float searchTrimDist;
	};

// vertex.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>		// for getch()
#include <alloc.h>		// for coreleft()
#include <stdlib.h>		// for itoa()
#include <string.h>		// for strcpy()
#include "edge.hpp"
#include "vertex.hpp"

VertexList::VertexList()
	{
	vertexHead = (VERTEX_NODE*)NULL;
   searchHead = (VERTEX_NODE*)NULL;
   searchMarker = 0;
   searchTrimDist = -1.0;
   cout << "Initialised Vertex class\n";
	}



VertexList::~VertexList()
	{
	cout << "VertexList destructor started\n";
	DelAllVertices();
	cout << "VertexList destructor ended!\n";
	}


void VertexList::AddToSearchList(VERTEX_NODE*a)
	{
	VERTEX_NODE *tmp;

	if (a == (VERTEX_NODE*)NULL)
		return;
```

```
	if (((*a).searchPrev != (VERTEX_NODE*)NULL) // if already in fringe
	|| ((*a).searchNext != (VERTEX_NODE*)NULL)) // list, dont add again
		return;

	if (searchHead == (VERTEX_NODE*)NULL)
		{
		searchHead = a;
		a->pathFrom = a->pathTo = (VERTEX_NODE*)NULL;
		a->searchPrev = a->searchNext = a;

//		cout << "Inserted into empty search list (";
//		cout << a->t << "," << a->w << "," << a->l << ")\n";
		}
	else
		{
		a->searchNext = searchHead;
		a->searchPrev = searchHead->searchPrev;
		searchHead->searchPrev = a;
		tmp = a->searchPrev;
		tmp->searchNext = a;
		searchHead = a;
//		cout << "Appended to search list (";
//		cout << a->t << "," << a->w << "," << a->l << ")\n";
		}
	}


VERTEX_NODE* VertexList::BuildNewVertex(int t, int w, int l, char nodeType)
	{
	VERTEX_NODE *newPtr;

	newPtr = (VERTEX_NODE*)farcalloc(1, sizeof(VERTEX_NODE));
	if (newPtr == (VERTEX_NODE*)NULL)
		{
		cout << "Farcalloc() failed in BuildNewVertex!\n";
	getch();
		exit(0);
		}
	(*newPtr).t = t;
	(*newPtr).w = w;
	(*newPtr).l = l;
	(*newPtr).nodeType = nodeType;
	(*newPtr).edgeList = new EdgeList();
	(*newPtr).pathTo = (*newPtr).pathFrom = (VERTEX_NODE*)NULL;
	(*newPtr).searchDist = 0.0;
	(*newPtr).searchPrev = (*newPtr).searchNext = (VERTEX_NODE*)NULL;
	(*newPtr).prev = (*newPtr).next = (VERTEX_NODE*)NULL;
	if (nodeType == START)
		{
		(*newPtr).searchMarker = searchMarker+1;
		AddToSearchList(newPtr);
		}
	else
		(*newPtr).searchMarker = searchMarker;
	return(newPtr);
	}


void VertexList::DelAllVertices()
	{
	while (vertexHead != (VERTEX_NODE*)NULL)
		{
//		cout << "About to delete (" << vertexHead->t << "," << vertexHead->w ;
//		cout << "," << vertexHead->l << ") at addr: " << vertexHead ;
//		cout << ": Prev:" << vertexHead->prev << ": Next:" << vertexHead->next <<"\n";
		DelVertex(vertexHead);
		}
	cout << "----------\n";
//	getch();
	}


void VertexList::DelVertex(VERTEX_NODE *todie)
	{
	VERTEX_NODE *tmp;
	EDGE_NODE*a;

	if (todie == (VERTEX_NODE*)NULL)
		return;

	for(a = todie->edgeList->GetFirstEdge();
		a != (EDGE_NODE*)NULL;
```

```
            a = todie->edgeList->GetFirstEdge()
            {
        todie->edgeList->DelEdgeToVertex(a->t, a->w, a->l);
        tmp = FindVertex(a->t, a->w, a->l);
        if (tmp != (VERTEX_NODE*)NULL)
            {
            tmp->edgeList->DelEdgeToVertex(todie->t, todie->w, todie->l);
            }
//              tmp = (VERTEX_NODE*)(a->otherVertex);
//              cout << "On vertex (" << todie->t << "," << todie->w << ",";
//              cout << todie->l << "):Del edge to (" << tmp->t << "," << tmp->w << ",";
//              cout << tmp->l << ")\n";
//              cout << "todie at:" << todie << ". a at:" << a << "\n";
//              cout << "  tmp at:" << tmp << "\n";
//              todie->edgeList->DelEdgeToVertex((void*)tmp);
//              cout << "On vertex (" << tmp->t << "," << tmp->w << ",";
//              cout << tmp->l << "):Del edge to (" << todie->t << "," << todie->w << ",";
//              cout << todie->l << ")\n";
//              tmp->edgeList->DelEdgeToVertex((void*)todie);
            }
        delete todie->edgeList;
        RemoveFromSearchList(todie);

        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
            {
            vertexHead = (VERTEX_NODE*)NULL;
            free(todie);
            return;
            }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
            {
            tmp = (*todie).prev;
            (*tmp).next = (*todie).next;
            tmp = (*todie).next;
            (*tmp).prev = (*todie).prev;
            free(todie);
            return;
            }
        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
            {
            vertexHead = tmp = (*todie).next;
            (*tmp).prev = (VERTEX_NODE*)NULL;
            free(todie);
            return;
            }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
            {
            tmp = (*todie).prev;
            (*tmp).next = (VERTEX_NODE*)NULL;
            free(todie);
            return;
            }
        }


int VertexList::FindPath()
    {
        EDGE_NODE *e;
        int goalFound=FALSE;
        float dist;
        VERTEX_NODE *cur, *adj, *t1, *t2;

        if (searchHead->nodeType == START)
            searchMarker = searchHead->searchMarker;
        else
            return(FALSE);

        t1 = GetStartVertex();
        t2 = GetGoalVertex();
        if ((t1 != (VERTEX_NODE*)NULL)
        && (t2 != (VERTEX_NODE*)NULL)
        && (t1->w == t2->w)
        && (t1->l == t2->l))
            {
            t1->pathTo = t2;
            return(TRUE);
            }

        for (cur = searchHead; cur != (VERTEX_NODE*)NULL; cur = searchHead)
            {
```

```
            if ((cur->searchDist >= searchTrimDist)
            && (searchTrimDist > 0.0))
                    {
                    RemoveFromSearchList(cur);
                    continue;
                    }
            for(e = cur->edgeList->GetFirstEdge();
                    e != (EDGE_NODE*)NULL;
                    e = cur->edgeList->GetNextEdge(e))
                    {
        adj = FindVertex(e->t, e->w, e->l);
        if (adj == (VERTEX_NODE*)NULL)
           continue;

//        adj = (VERTEX_NODE*) e->otherVertex;
                    dist = cur->searchDist + e->dist;
                    if ( (adj->searchMarker != searchMarker)
                    || ((adj->searchMarker == searchMarker)
                           && (adj->searchDist > dist)) )
                           {
                           if ((dist <= searchTrimDist)
                           || (searchTrimDist <= 0.0))
                                  {
                                  adj->pathFrom = cur;
                                  adj->searchMarker = searchMarker;
                                  adj->searchDist = dist;
                                  AddToSearchList(adj);
                                  if (adj->nodeType == GOAL)
                                         {
                                         goalFound = TRUE;
                                         searchTrimDist = dist;
                                         MarkPath(adj);
                                         TrimSearchList();
                                         }

                           }
                    }
            RemoveFromSearchList(cur);
            }
      return(goalFound);
      }


VERTEX_NODE* VertexList::FindVertex(int t, int w, int l)
   {
   VERTEX_NODE* cur;

   for(cur = GetFirstVertex();
      cur != (VERTEX_NODE*)NULL;
      cur = GetNextVertex(cur))
      {
      if ((cur->t == t)
      && (cur->w == w)
      && (cur->l == l))
         return(cur);            // found it

      if ((cur->t >= t)
      && (cur->w >= w)
      && (cur->l >= l))
         break;                  // passed it - it's not in the list
         }
   return((VERTEX_NODE*)NULL);
   }


VERTEX_NODE* VertexList::GetFirstVertex()
   {
   return(vertexHead);
   }


VERTEX_NODE* VertexList::GetGoalVertex()
   {
   VERTEX_NODE*tmp;

   for(tmp = GetFirstVertex();
      tmp != (VERTEX_NODE*)NULL;
      tmp = GetNextVertex(tmp))
      {
            if (tmp->nodeType == GOAL)
         break;
      }
   return(tmp);
```

```
    }

VERTEX_NODE* VertexList::GetNextVertex(VERTEX_NODE*cur)
    {
    return(cur->next);
    }

int VertexList::GetSearchMarker()
        {
        return(searchMarker);
        }

VERTEX_NODE* VertexList::GetStartVertex()
        {
        VERTEX_NODE*tmp;

        for(tmp = GetFirstVertex();
                tmp != (VERTEX_NODE*)NULL;
                tmp = GetNextVertex(tmp))
                {
                if (tmp->nodeType == START)
          break;
      }
   return(tmp);
   }

void VertexList::InsertNewVertex(VERTEX_NODE*a)
        {
        VERTEX_NODE *cur;

        if (a == (VERTEX_NODE*)NULL)
                return;

        if (vertexHead == (VERTEX_NODE*)NULL)
                {
                vertexHead = a;
                (*a).prev = (*a).next = (VERTEX_NODE*)NULL;
                return;
                }

        for(cur=vertexHead; cur != (VERTEX_NODE*)NULL;   cur=(*cur).next)
                {
                if ( (a->t > cur->t)
       || ((a->t == cur->t)
          && (a->w > cur->w))
        || ((a->t == cur->t)
          && (a->w == cur->w)
                && (a->l > cur->l)) )
                        {                                   // insert after cur
                        if ((*cur).next == (VERTEX_NODE*)NULL)
                                {
                                (*a).next = (*cur).next;     // eol - append new node
                                (*a).prev = cur;
                                (*cur).next = a;
                                break;
                                }
                        else
                                continue;                    // get next node
                        }
                if ( (a->t < cur->t)
                || ((a->t == cur->t)
                        && (a->w < cur->w))
                || ((a->t == cur->t)
                        && (a->w == cur->w)
                        && (a->l < cur->l)) )
                        {                                   // insert before cur
                        if (cur->prev == (VERTEX_NODE*)NULL)
                                {
                                a->prev = (VERTEX_NODE*)NULL;// at start of list
                                a->next = cur;
                                cur->prev = a;
                                vertexHead = a;
                                break;
                                }
                        else
                                {
                                a->prev = cur->prev;          // in middle/end of list
                                a->next = cur;
                                cur->prev = a;
                                cur = a->prev;
```

```
                        cur->next = a;
                        break;
                        }
                }
        if ((a->t == cur->t)
        && (a->w == cur->w)
        && (a->l == cur->l))
                {                                       // insert after cur at eol
                if ((*cur).next == (VERTEX_NODE*)NULL)
                        {
                        (*a).next = (VERTEX_NODE*)NULL;
                        (*a).prev = cur;
                        (*cur).next = a;
                        break;
                        }
                else
                        continue;
                }
        }
//      AddToSearchList(a);
//      cout << "Inserted vertex (";
// cout << (*a).t << "," << (*a).w << "," << (*a).l << ")\n";
        }


void VertexList::ListAllVertices(int debugFlag)
        {
        char tmp[256];
        fstream debugFile;
        VERTEX_NODE *cur;

        strcpy(tmp, "List of all nodes in list\n");
        cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        for(cur=vertexHead; cur != (VERTEX_NODE*)NULL; cur=cur->next)
                {
                sprintf(tmp, "Vertex:%c: at (%d,%d,%d)\n",
                                (*cur).nodeType, (*cur).t, (*cur).w, (*cur).l);
                cout << tmp;
                if (debugFlag)
                        {
                        debugFile.open(DEBUG_FILENAME, ios::app);
                        debugFile.write(tmp, strlen(tmp));
                        debugFile.close();
                        }
                cur->edgeList->ListAllEdges(debugFlag);
                }
        }


void VertexList::ListSearchList(int debugFlag)
        {
        char tmp[256];
        fstream debugFile;
        VERTEX_NODE*cur;

        if (searchHead == (VERTEX_NODE*)NULL)
                {
                sprintf(tmp,"Empty SearchList! (marker=%d)\n", searchMarker);
      cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
                return;
                }

        sprintf(tmp, "List of nodes in Search list (marker=%d)\n", searchMarker);
   cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        cur = searchHead;
        do
```

```
		{
		sprintf(tmp, "SearchList node (%d,%d,%d) Dist = %f",
		cur->t, cur->w, cur->l, cur->searchDist);
	cout << tmp;
	if (debugFlag)
		{
		debugFile.open(DEBUG_FILENAME, ios::app);
		debugFile.write(tmp, strlen(tmp));
		debugFile.close();
		}
		cur=cur->searchNext;
		}
	while (cur != searchHead);
    strcpy(tmp, "End of SearchList\n");
    cout << tmp;
	if (debugFlag)
		{
		debugFile.open(DEBUG_FILENAME, ios::app);
		debugFile.write(tmp, strlen(tmp));
		debugFile.close();
		}
	}


void VertexList::MarkPath(VERTEX_NODE*v)
	{
	VERTEX_NODE*tmp;

	if (v == (VERTEX_NODE*)NULL)
		return;

//	cout << "Path from Goal to Start\n";
	v->pathTo = (VERTEX_NODE*)NULL;
	do
		{
		tmp = v->pathFrom;
//		cout << "(" << v->t << "," << v->w << "," << v->l << ")\n";
		if (tmp != (VERTEX_NODE*)NULL)
			{
			tmp->pathTo = v;
			v = v->pathFrom;
			}
		}
	while (tmp != (VERTEX_NODE*)NULL);
	}


void VertexList::RemoveFromSearchList(VERTEX_NODE*a)
	{
	VERTEX_NODE *cur;

	if ((a->searchPrev == (VERTEX_NODE*)NULL)
	|| (a->searchNext == (VERTEX_NODE*)NULL))
	{
	if (searchHead == a)
	   searchHead = (VERTEX_NODE*)NULL;
	return;
	}

	cur=a->searchPrev;
	if (cur == a)
		{
	searchHead = (VERTEX_NODE*)NULL;
//		cout << "Removed Search Node (";
//	cout << a->t << "," << a->w << "," << a->l << "). SearchList empty\n";
		}
	else
		{
		cur->searchNext = a->searchNext;
		cur = a->searchNext;
		cur->searchPrev = a->searchPrev;
		if (searchHead == a)
			searchHead = a->searchNext;
//		cout << "Removed Search Node (";
//	cout << a->t << "," << a->w << "," << a->l << ").\n";
		}
	a->searchPrev = a->searchNext = (VERTEX_NODE*)NULL;
	}


void VertexList::RemoveVertex(VERTEX_NODE *todie)
	{
	VERTEX_NODE *tmp;
```

```
	if (todie == (VERTEX_NODE*)NULL)
		return;

	RemoveFromSearchList(todie);

	if ( ((*todie).prev == (VERTEX_NODE*)NULL)
	&& ((*todie).next == (VERTEX_NODE*)NULL) )
		{
		vertexHead = (VERTEX_NODE*)NULL;
		return;
		}
	if ( ((*todie).prev != (VERTEX_NODE*)NULL)
	&& ((*todie).next != (VERTEX_NODE*)NULL) )
		{
		tmp = (*todie).prev;
		(*tmp).next = (*todie).next;
		tmp = (*todie).next;
		(*tmp).prev = (*todie).prev;
		return;
		}
	if ( ((*todie).prev == (VERTEX_NODE*)NULL)
	&& ((*todie).next != (VERTEX_NODE*)NULL) )
		{
		vertexHead = tmp = (*todie).next;
		(*tmp).prev = (VERTEX_NODE*)NULL;
		return;
		}
	if ( ((*todie).prev != (VERTEX_NODE*)NULL)
	&& ((*todie).next == (VERTEX_NODE*)NULL) )
		{
		tmp = (*todie).prev;
		(*tmp).next = (VERTEX_NODE*)NULL;
		return;
		}
	}


void VertexList::TrimSearchList()
   {
	VERTEX_NODE *cur, *tmpPtr=(VERTEX_NODE*)0;
	int trimmed=TRUE;

//	cout << "Trimming search list to " << searchTrimDist << " or less\n";
	if ((searchHead == (VERTEX_NODE*)NULL)
   || (searchTrimDist < 0.0))
		return;

   cur = searchHead;
	while ((tmpPtr != cur) || (trimmed == TRUE))
		{
		if (trimmed==TRUE)
			{
			tmpPtr=cur;
			trimmed=FALSE;
			}
		if ((*cur).searchDist >= searchTrimDist)
			{
//			cout << "Trimmed out (";
//		cout << cur->t << "," << cur->w << "," << cur->l << ") ";
//		cout << "Dist was " << cur->searchDist << "\n";
			if (cur == cur->searchNext)
				{
		 RemoveFromSearchList(cur);
				break;
				}
			else
				{
				tmpPtr = cur->searchNext;
				RemoveFromSearchList(cur);
				cur = tmpPtr;
				trimmed = TRUE;
				}
			}
		else
			cur = cur->searchNext;
		}
//    cout << "-------\n";
   }
```

B.9. Partial Update Priority First Graph Theory

The Partial Update Priority First Search Algorithm for Graph Theory which was developed and implemented as part of this project was coded in ANSI C++. The filenames for each of the separate source code files were supplied inside C++ format comments (i.e. //) at the beginning of each file listing. A detailed explanation of the design behind this program was presented in Chapter 4.

```
// main.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <sys\timeb.h>
#include <dir.h>
#include <ctype.h>
#include "edge.hpp"
#include "vertex.hpp"
#include "object.hpp"
#include "domain.hpp"
#include "bench.hpp"

#define MAPFILE_MASK "MAP*.DAT"

int main(int, char**);
void BuildUpdateListFromDomain(VertexList&, VertexList&, Domain&);
void BuildVertexListFromDomain(VertexList&, Domain&);
void FindAllEdges(VertexList&, Domain&);
void FindEdgesForUpdateNode(VERTEX_NODE*, VertexList&, VertexList&, Domain&);
int MemStatus(char*);
void MergeLists(VertexList&, VertexList&, Domain&);
int SetWorkingDir(void);
void RenameMapFile(char*);


int main(int argc, char** argv)
	{
	char* shortName;
	struct ffblk mapfile;

	for(;;)
		{
		shortName = strstr(argv[0], "\\");
		if (shortName == (char*)NULL)
			{
			shortName = argv[0];
			cout << "This is " << shortName << "\n";
			break;
			}
		else
			argv[0] = shortName+1;
		}

	MemStatus("Free memory before mainloop in main: ");

	if (!SetWorkingDir())
		{
		cout << "Program exiting gracefully\n";
		return(1);
		}
	if (findfirst(MAPFILE_MASK, &mapfile, 0))
		{
		cout << "Program exiting gracefully - no map files found\n";
		return(1);
		}
	do
		{
		char tmp[256];
		int i;
		Benchmark stopWatch(shortName);

		sprintf(tmp, "Starting on %s: ", mapfile.ff_name);
		MemStatus(tmp);
		for (i=0; i<NUMTIMES; i++)
			{
			int timeTaken;
			float distTravelled;
			Domain theWorld(10, 20, 20, mapfile.ff_name);
			VertexList vertList, updateList;
```

```
                    timeTaken = 0;
                    distTravelled = 0.0;
                    stopWatch.IterStart(i, mapfile.ff_name);
                    stopWatch.Click();
                    BuildVertexListFromDomain(vertList, theWorld);
                    FindAllEdges(vertList, theWorld);
                    for(;;)
                            {
                            int pathFoundFlag, goalFoundFlag, a, b, c;
                            float dist;
                            VERTEX_NODE *t1;

                            stopWatch.Click();
                            theWorld.DrawDomain();
                            stopWatch.Click();
                            pathFoundFlag = vertList.FindPath();
                            stopWatch.Click();

                            if (pathFoundFlag == TRUE)
                                    {
                                    cout << "Path found to the GOAL!";
                                    cout << "(dist=" << distTravelled << ")\n";
                                    }
                            else
                                    {
                                    cout << "No path found to the GOAL!";
                                    cout << "(dist=" << distTravelled << ")\n";
                                    }

                            t1 = vertList.GetStartVertex();
                            if (pathFoundFlag)
                                    {
                                    a = t1->pathTo->t;
                                    b = t1->pathTo->w;
                                    c = t1->pathTo->l;
                                    }
                            else
                                    {
                                    a = (t1->t)+1;
                                    b = t1->w;
                                    c = t1->l;
                                    }
                            goalFoundFlag = theWorld.MoveRobot(t1->t, t1->w, t1->l,

a, b, c, &dist);
                            distTravelled += dist;
                            timeTaken++;
                            if (goalFoundFlag)
                                    {
                                    cout << "MADE IT TO THE GOAL!\n";
                                    break;
                                    }

                            theWorld.AdvanceTime();
                            stopWatch.Click();
                            BuildUpdateListFromDomain(vertList, updateList, theWorld);
                            MergeLists(vertList, updateList, theWorld);
                            }
                    stopWatch.IterStop(timeTaken, distTravelled);
                    }
            stopWatch.LogCalcs();
            RenameMapFile(mapfile.ff_name);
            }
        while (!findnext(&mapfile));

        MemStatus("Free memory after mainloop in main: ");
        return(0);
        }


void BuildUpdateListFromDomain(VertexList &vList, VertexList &uList, Domain &domain)
        {
        char type;
        int t=domain.GetDomainTimeSlices(), a;
        int w=domain.GetDomainWidth(), b;
        int l=domain.GetDomainLength(), c;
        VERTEX_NODE*newPtr;

        for(a=0; a<t; a++)
          {
            for(b=0; b<w; b++)
                {
```

```
                    for(c=0; c<l; c++)
                        {
                        type = domain.GetPointType(a, b, c);
                        if ((type == VERTEX)
                        || (type == START)
                        || (type == GOAL))
                            {
                            VERTEX_NODE *tmpv, *tmpu;

                            tmpv = vList.FindVertex(a, b, c);
                            tmpu = uList.FindVertex(a, b, c);

                            if ((!tmpv)
                            && (!tmpu))
                                {
                                newPtr = uList.BuildNewVertex(a, b, c, type);
                                uList.InsertNewVertex(newPtr);
                                continue;
                                }

                            if ((tmpv != (VERTEX_NODE*)NULL)
                            && (tmpv->nodeType != type))
                                {
                                vList.RemoveVertex(tmpv);
                                vList.DelVertex(tmpv);
                                newPtr = uList.BuildNewVertex(a, b, c, type);
                                uList.InsertNewVertex(newPtr);
                                continue;
                                }

                            if ((tmpu != (VERTEX_NODE*)NULL)
                            && (tmpu->nodeType != type))
                                {
                                uList.RemoveVertex(tmpu);
                                uList.DelVertex(tmpu);
                                newPtr = uList.BuildNewVertex(a, b, c, type);
                                uList.InsertNewVertex(newPtr);
                                continue;
                                }
                            }
                        else
                            {
                            VERTEX_NODE *v;

                            v = vList.FindVertex(a, b, c);
                            if (v!=(VERTEX_NODE*)NULL)
                                {
                                EDGE_NODE*e;

                                for(e = v->edgeList->GetFirstEdge();
                                    e != (EDGE_NODE*)NULL;
                                    e = v->edgeList->GetNextEdge(e))
                                    {
                                    VERTEX_NODE* v2;

                                    v2 = vList.FindVertex(e->t, e->w, e->l);
                                    if (v2 != (VERTEX_NODE*)NULL)
                                        {
                                        vList.RemoveVertex(v2);
                                        uList.InsertNewVertex(v2);
                                        v2->edgeList->DelAllEdges();
                                        }
                                    }
                                vList.DelVertex(v);
                                }
                            v = uList.FindVertex(a, b, c);
                            if (v != (VERTEX_NODE*)NULL)
                                {
                                uList.DelVertex(v);
                                }
                            }
                        }
                    }
                }
        }


/*void BuildVertexListFromDomain(VertexList &vList, Domain &domain)
    {
        char type;
        int t=domain.GetDomainTimeSlices(), a;
        int w=domain.GetDomainWidth(), b;
        int l=domain.GetDomainLength(), c;
        VERTEX_NODE*newPtr;
```

```
        for(a=0; a<t; a++)
          {
                for(b=0; b<w; b++)
                        {
                        for(c=0; c<l; c++)
                                {
                                type = domain.GetPointType(a, b, c);
                                if ((type == VERTEX)
                                || (type == START)
                                || (type == GOAL))
                                        {
                                newPtr = vList.BuildNewVertex(a, b, c, type);
                                vList.InsertNewVertex(newPtr);

                                }
                        }
                }
        }*/


void BuildVertexListFromDomain(VertexList &vList, Domain &domain)
        {
        char type;
        int t=domain.GetDomainTimeSlices(), a;
        int w=domain.GetDomainWidth(), b;
        int l=domain.GetDomainLength(), c;
        VERTEX_NODE*newPtr, *gPtr, *sPtr;

        for(gPtr = sPtr = (VERTEX_NODE*)NULL, a=0; a<t; a++)
          {
                for(b=0; b<w; b++)
                        {
                        for(c=0; c<l; c++)
                                {
                                type = domain.GetPointType(a, b, c);
                                if ((type == VERTEX)
                                || (type == START)
                                || (type == GOAL))
                                        {
                                        newPtr = vList.BuildNewVertex(a, b, c, type);
                                        vList.InsertNewVertex(newPtr);
                                        }
              if (type == GOAL)
                 gPtr = newPtr;
              if (type == START)
                 sPtr = newPtr;
                                }
                        }
                }
    if ((gPtr != (VERTEX_NODE*)NULL)
    && (sPtr != (VERTEX_NODE*)NULL))
       {
       sPtr->searchDist = 0.0;
                vList.AddToSearchList(sPtr);
       }
        }



void FindAllEdges(VertexList &vList, Domain &domain)
        {
        EDGE_NODE*e;
        VERTEX_NODE*a,*b;
        float dist;

        for(a = vList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = vList.GetNextVertex(a))
                {
                for(b = vList.GetNextVertex(a);
                        b != (VERTEX_NODE*)NULL;
                        b = vList.GetNextVertex(b))
          {
                        if (a == b)
                                continue;

                        if ((a->t == b->t)
                        && (a->w == b->w)
                        && (a->l == b->l))
                                continue;

          dist = domain.CheckLine(a->t, a->w, a->l, b->t, b->w, b->l);
```

```
        if (dist > 0.0)
            {
                            e = a->edgeList->BuildNewEdge(b->t, b->w, b->l, dist);
                            a->edgeList->InsertNewEdge(e);
                            }
                dist = domain.CheckLine(b->t, b->w, b->l, a->t, a->w, a->l);
                if (dist > 0.0)
                            {
                            e = b->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                            b->edgeList->InsertNewEdge(e);
                            }
                }
            }
        }


void FindEdgesForUpdateNode(VERTEX_NODE*cur, VertexList &vList, VertexList &uList, Domain &domain)
        {
        EDGE_NODE*e;
        VERTEX_NODE*a;
        float dist;

        for(a = vList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = vList.GetNextVertex(a))
                {
        if (a == cur)
                continue;

                if ((a->t == cur->t)
                && (a->w == cur->w)
                && (a->l == cur->l))
                        continue;

        dist = domain.CheckLine(a->t, a->w, a->l, cur->t, cur->w, cur->l);
        if (dist > 0.0)
           {
                        e = a->edgeList->BuildNewEdge(cur->t, cur->w, cur->l, dist);
                        a->edgeList->InsertNewEdge(e);
                        }
                dist = domain.CheckLine(cur->t, cur->w, cur->l, a->t, a->w, a->l);
                if (dist > 0.0)
                        {
                        e = cur->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                        cur->edgeList->InsertNewEdge(e);
                        }
                }

        for(a = uList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = uList.GetNextVertex(a))
                {
        if (a == cur)
                continue;

                if ((a->t == cur->t)
                && (a->w == cur->w)
                && (a->l == cur->l))
                        continue;

        dist = domain.CheckLine(a->t, a->w, a->l, cur->t, cur->w, cur->l);
        if (dist > 0.0)
                        {
                        e = a->edgeList->BuildNewEdge(cur->t, cur->w, cur->l, dist);
                        a->edgeList->InsertNewEdge(e);
                        }
                dist = domain.CheckLine(cur->t, cur->w, cur->l, a->t, a->w, a->l);
                if (dist > 0.0)
                        {
                        e = cur->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                        cur->edgeList->InsertNewEdge(e);
           }
                }
        }


int MemStatus(char *StatusMessage)
        {
        char tmp[256];
        fstream debugFile;
        long MemLeft;
        int ret;

        debugFile.open("DEBUG.LOG", ios::app);
```

```
	if (!debugFile)
		cout << "Unable to open DEBUG.LOG\n";

	MemLeft = (long) coreleft();
	sprintf(tmp, "%s%ld\n", StatusMessage, MemLeft);
	debugFile.write(tmp, strlen(tmp));
	debugFile.close();
//	cout << StatusMessage << MemLeft << "\n";

	ret = farheapcheck();
	if (ret == _HEAPOK)
		cout << "Heap ok" << StatusMessage << "\n";
	else
		{
		cout << "Heap error <" << ret << ">" << StatusMessage<< "\n";
		getch();
		return(FALSE);
		}
	return(TRUE);
	}


void MergeLists(VertexList &vList, VertexList &uList, Domain &domain)
   {
   VERTEX_NODE* cur;

   for (cur=uList.GetFirstVertex();
       cur!=(VERTEX_NODE*)NULL;
       cur=uList.GetFirstVertex())
       {
       uList.RemoveVertex(cur);
		FindEdgesForUpdateNode(cur, vList, uList, domain);
		vList.InsertNewVertex(cur);
		}
	cur = vList.GetStartVertex();
	if (cur != (VERTEX_NODE*)NULL)
		{
		cur->searchMarker = vList.GetSearchMarker() + 1;
		vList.AddToSearchList(cur);
		}
	}


int SetWorkingDir()
	{
	char mapdir[256];

	cout <<"Enter the directory containing map files, or \"q\" for quit:";
	cin >> mapdir;
	if (toupper(mapdir[0]) == 'Q')
		{
		cout << "Quitting...\n";
		return(FALSE);
		}
	if (chdir(mapdir))
		{
		cout << "The directory " << mapdir << " could not be found.\n";
		return(FALSE);
		}
	cout << "Made " << mapdir << " the current directory.\n";
	return(TRUE);
	}


void RenameMapFile(char*filename)
	{
	char newfilename[128];
	char* ch;
	int i=(int)'.';

	strcpy(newfilename, filename);
	ch = strrchr(newfilename,i);
	if (ch != (char*)NULL)
		{
		strcpy(ch, ".bak");
		rename(filename, newfilename);
		}
	else
		exit(1);
	}

// bench.hpp
```

```
#define FALSE 0
#define TRUE !FALSE
#define NUMTIMES 10
#define MAXFILENAME 13

class Benchmark
	{
	public:
		Benchmark(char*);
		~Benchmark();
		void Click(void);
		void IterStart(int, char*);
		void IterStop(int, float);
		void LogCalcs(void);
	private:
		void Diff(struct timeb*, struct timeb*, struct timeb*);

		fstream logFile;
		fstream avgFile;
		char mapFileName[MAXFILENAME];
		struct timeb benchmarks[NUMTIMES][2];
		float distRobotTravelled[NUMTIMES];
		int timeTaken[NUMTIMES];
		int currentIter;
		int clickToggleFlag;
		struct timeb elapsedTime, computeTime;
		struct timeb startTime, clickOnTime, clickOffTime;
	};

// bench.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <alloc.h>		// for coreleft()
#include <stdlib.h>		// for itoa()
#include <string.h>		// for strcpy()
#include <sys\timeb.h>
#include "bench.hpp"


Benchmark::Benchmark(char *fileName)
	{
	char *tmp;
	char logname[MAXFILENAME];

	while((tmp = strchr(fileName, '\\')) != (char*)NULL)
		fileName = tmp+1;
	strcpy(logname, fileName);
	if ((tmp = strchr(logname, '.')) != (char*)NULL)
		strcpy(tmp, ".LOG");
	else
		strcat(logname, ".LOG");
	logFile.open(logname, ios::app);
	if (!logFile)
		cout << "Unable to open " << logname << "\n";

	if ((tmp = strchr(logname, '.')) != (char*)NULL)
		strcpy(tmp, ".AVG");
	else
		strcat(logname, ".AVG");
	avgFile.open(logname, ios::app);
	if (!avgFile)
		cout << "Unable to open " << logname << "\n";

	cout << "Initialised Benchmark class\n";
	}


Benchmark::~Benchmark()
	{
	if (logFile)
		{
		logFile.flush();
		logFile.close();
		}
	if (avgFile)
		{
		avgFile.flush();
		avgFile.close();
		}
	cout << "Closed log files and Destroying Benchmark class\n";
	}
```

```
void Benchmark::Click()
	{
	switch(clickToggleFlag)
		{
		case TRUE:
			ftime(&clickOnTime);
			clickToggleFlag = FALSE;
			break;
		case FALSE:
		default:
			ftime(&clickOffTime);
			Diff(&clickOnTime, &clickOffTime, &computeTime);
			clickToggleFlag = TRUE;
		}
	}


void Benchmark::Diff(struct timeb*start, struct timeb*stop, struct timeb*diff)
	{
	if ((*stop).millitm < (*start).millitm)
		{
		(*stop).millitm += (short)1000; /* carry when subtracting, stops*/
		(*start).time += 1L;            /* negative wraparound problems!*/
		}
	(*diff).millitm += (*stop).millitm - (*start).millitm;
	(*diff).time += (long)((*diff).millitm / (short)1000);
	(*diff).millitm %= (short)1000;
	(*diff).time += ((*stop).time-(*start).time);
	}


void Benchmark::IterStart(int i, char*s)
	{
	currentIter = i;
	clickToggleFlag = TRUE;
	computeTime.time = elapsedTime.time = 0L;
	computeTime.millitm = elapsedTime.millitm = 0;

	strcpy(mapFileName, s);

	ftime(&startTime);
	}


void Benchmark::IterStop(int t, float distTravelled)
	{
	struct timeb stopTime;

	ftime(&stopTime);
	Diff(&startTime, &stopTime, &elapsedTime);

	benchmarks[currentIter][0].time = elapsedTime.time;
	benchmarks[currentIter][0].millitm = elapsedTime.millitm;
	benchmarks[currentIter][1].time = computeTime.time;
	benchmarks[currentIter][1].millitm = computeTime.millitm;
	timeTaken[currentIter] = t;
	distRobotTravelled[currentIter] = distTravelled;
	}



void Benchmark::LogCalcs()
	{
	char tmp[256];
	float avgDist;
	struct timeb avg;
	int i, avgTimeTaken;

	for(i=0, avg.time=0L, avg.millitm=0, avgDist=0.0, avgTimeTaken=0;
		i<NUMTIMES;
		i++)
		{
		sprintf(tmp, "%s (%02d) Elapsed time:%05ld.%03d\n", mapFileName, i,
				benchmarks[i][0].time, benchmarks[i][0].millitm);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Compute time:%05ld.%03d\n", mapFileName, i,
				benchmarks[i][1].time, benchmarks[i][1].millitm);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Dist travelled:%f\n", mapFileName, i,
				distRobotTravelled[i]);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Time Slices Taken:%d\n", mapFileName, i,
				timeTaken[i]);
```

```
                logFile.write(tmp, strlen(tmp));
                avg.time+=benchmarks[i][0].time;
                avg.millitm+=benchmarks[i][0].millitm;
                if (avg.millitm % 1000 != avg.millitm)
                        {
                        avg.time += (long)(avg.millitm / (short)1000);
                        avg.millitm %= (short)1000;
                        }
                }

        sprintf(tmp, "%s Tot. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));
        cout << avg.time << "." << avg.millitm << "\n";
        cout << tmp << "\n";

        i = (int) (avg.time % (long)NUMTIMES);
        avg.time /= (long)NUMTIMES;
        avg.millitm += i * 1000;
        avg.millitm /= NUMTIMES;

        sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));

        for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
                {
                avg.time+=benchmarks[i][1].time;
                avg.millitm+=benchmarks[i][1].millitm;
                if (avg.millitm % 1000 != avg.millitm)
                        {
                        avg.time += (long)(avg.millitm / (short)1000);
                        avg.millitm %= (short)1000;
                        }
                }
        sprintf(tmp, "%s Tot. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));
        i = (int) (avg.time % (long)NUMTIMES);
        avg.time /= (long)NUMTIMES;
        avg.millitm += i * 1000;
        avg.millitm /= NUMTIMES;
        sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
        logFile.write(tmp, strlen(tmp));

        for(i=0, avgTimeTaken=0; i<NUMTIMES; i++)
                avgTimeTaken+=timeTaken[i];
        avgTimeTaken = avgTimeTaken / NUMTIMES;
        sprintf(tmp, "%s Avg. Time Taken:%d\n", mapFileName, avgTimeTaken);
        avgFile.write(tmp, strlen(tmp));
        logFile.write(tmp, strlen(tmp));

        for(i=0, avgDist=0.0; i<NUMTIMES; i++)
                avgDist += distRobotTravelled[i];
        avgDist = avgDist / ((float)NUMTIMES);
        sprintf(tmp, "%s Avg. Dist Travelled:%f\n", mapFileName, avgDist);
        avgFile.write(tmp, strlen(tmp));
        logFile.write(tmp, strlen(tmp));

        sprintf(tmp, "%s ====================\n", mapFileName);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s ====================\n", mapFileName);
        logFile.write(tmp, strlen(tmp));
        }

// domain.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00
```

```
#define FROM_FRONT 0
#define FROM_BACK 1
#define FROM_LEFT 2
#define FROM_RIGHT 3
#define FROM_FRONTRIGHT 4
#define FROM_FRONTLEFT 5
#define FROM_BACKRIGHT 6
#define FROM_BACKLEFT 7
#define FROM_UP 8
#define FROM_DOWN 9
#define FROM_NOWHERE 10

#define NOCOST (float)0.0
#define NORMAL (float)1.0
#define NORMAL_DIAG (float)1.414214
#define BLOCKED (float)1000.0

typedef struct
       {
       float dist;
       int from;
       char type;
       float cost[9];
       } POINT;

class Domain
       {
       public:
              Domain(int, int, int, char*);
              ~Domain();
              void AdvanceTime(void);
       float CheckLine(int, int, int, int, int, int);
              void DrawDomain(void);
       int GetDomainLength(void) { return(domainLength); }
       int GetDomainTimeSlices(void) { return(domainTimeSlices); }
       int GetDomainWidth(void) { return(domainWidth); }
              POINT* GetPoint(int, int, int);
       float GetPointCost(int, int, int, int);
              char GetPointFrom(int, int, int);
              char GetPointType(int, int, int);
              int IsPointClear(int, int, int);
              int IsPointGoal(int, int, int);
              int IsPointNearObject(int, int, int);
              int IsPointObject(int, int, int);
              int IsPointStart(int, int, int);
       int IsPointVertex(int, int, int);
              int MoveRobot(int, int, int, int, int, int, float*);
              void SetPointFrom(int, int, int, int);
              void SetPointType(int, int, int, char);
       private:
              void AgeTimeSlices(void);
       int CalcRobotDir(int, int, int, int, int, int);
              int ClearAdjPointOK(int, int, int);
              void ClearMobileObject(int, int, int);
              void ClearVerticesInTimeSlice(int);
              void DrawTimeSlice(int);
              void InitTimeSlice(int);
              void MarkMobileObject(int, int, int);
              void MoveMobileObject(OBJECT_NODE*);
              void MoveMobileObjects(void);
              void SetAdjObjsInTimeSlice(int);
              void SetGoalFromFile(char*);
              void SetMobileObjsFromFile(char*);
              void SetPermObjsInTimeSlice(int);
              void SetStartFromFile(char*);
              void SetVerticesInTimeSlice(int);

              ObjectList objList;
              POINT** domainHead;
       int domainWidth;
       int domainLength;
       int domainTimeSlices;
       };

// domain.cpp
#include <iostream.h>
#include <fstream.h>
#include <alloc.h>          // for coreleft()
#include <stdio.h>
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <conio.h>
#include "object.hpp"
#include "domain.hpp"
```

```
Domain::Domain(int numTimeSlices, int width, int length, char*mapFileName)
	{
	int i;

	cout << "Contructing Domain class\n";
	domainTimeSlices = numTimeSlices;
	domainWidth = width;
	domainLength = length;

	cout << "FreeHeap:" << farcoreleft() << "\n";
	cout << "Amount needed for one timeslice:";
	cout << width*length*sizeof(POINT) << "\n";

	domainHead = (POINT**)farcalloc(numTimeSlices, sizeof(POINT**));
	if (domainHead == (POINT**)NULL)
		{
		cout << "Not enough memory to build model of world\n";
		domainHead = (POINT***)NULL;
		domainTimeSlices = domainWidth = domainLength = 0;
		return;
		}

	for(i=0; i<numTimeSlices; i++)
		{
		domainHead[i] = (POINT*)farcalloc(width*length, sizeof(POINT));
		cout << "FreeHeap after timeslice allocated:" << farcoreleft() << "\n";
		if (domainHead[i] == (POINT*)NULL)
			{
			if (i==0)
				cout << "Not enough memory to build model of world\n";
			else
				cout << "Only enough memory to build " << i << " of the ";
				cout << numTimeSlices << " timeslices in model of world\n";
			domainTimeSlices = i;
			break;
			}
		InitTimeSlice(i);
		SetPermObjsInTimeSlice(i);
		}
	SetStartFromFile(mapFileName);
	SetGoalFromFile(mapFileName);
	SetMobileObjsFromFile(mapFileName);
	for(i=0; i<domainTimeSlices; i++)
		{
		SetAdjObjsInTimeSlice(i);
		SetVerticesInTimeSlice(i);
		}
	for(i=0; i<domainTimeSlices-1; i++)
		AdvanceTime();
	cout << "FreeHeap after Domain allocated:" << farcoreleft() << "\n";
	}


Domain::~Domain()
	{
	POINT *tmp;
	int i;

	cout << "Destructing Domain class\n";
	if (domainHead == (POINT**)NULL)
		{
		cout << "Not freeing domain - DomainHead was NULL\n";
		return;
		}
	for(i=0; i<domainTimeSlices; i++)
		{
		tmp = domainHead[i];
		farfree(tmp);
		}
	farfree(domainHead);
	domainHead = (POINT**)NULL;
	cout << "FreeHeap after Domain deallocated:" << farcoreleft() << "\n";
//	getch();
	}


void Domain::AdvanceTime()
	{
	AgeTimeSlices();
//	DrawDomain();
	ClearVerticesInTimeSlice(domainTimeSlices-1);
	MoveMobileObjects();
```

```
//      DrawDomain();
        SetVerticesInTimeSlice(domainTimeSlices-1);
//      DrawDomain();
        }


////////////////////////////////////////
//
// Copy the contents of every timeslice
// into the previous timeslice
// [0] <= [1], [1] <= [2], etc.
// Leave the last timeslice unchanged.
// Another routine will decide the moves
// for all the mobile objects.
//
// A quick way to do this is moving ptrs
// to the timeslices and only copying the
// contents of the last timeslice over the
// contents of the first timeslice
//
////////////////////////////////////////
void Domain::AgeTimeSlices()
        {
        int i, j;
        POINT *tmp;

        tmp = domainHead[0];
        for(i=0, j=1; i<domainTimeSlices-1; i++, j++)
                domainHead[i] = domainHead[j];
        domainHead[domainTimeSlices-1] = tmp;
        _fmemcpy(domainHead[domainTimeSlices-1],
                                domainHead[domainTimeSlices-2],
                                sizeof(*domainHead[domainTimeSlices-1]));
        }


int Domain::CalcRobotDir(int st, int sw, int sl, int et, int ew, int el)
        {
        if ((!IsPointStart(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(clear);

        if ((ew - sw > 0)
        && (el - sl > 0))
                {
                sw++, sl++;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
                || (IsPointGoal(st, sw, sl)))
                        return(frontright);
                else
                        return(clear);
                }

        if ((ew - sw > 0)
        && (el - sl == 0))
                {
                sw++;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
                || (IsPointGoal(st, sw, sl)))
                        return(right);
                else
                        return(clear);
                }

        if ((ew - sw > 0)
        && (el - sl < 0))
                {
                sw++, sl--;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
                || (IsPointGoal(st, sw, sl)))
                        return(backright);
                else
                        return(clear);
                }

        if ((ew - sw == 0)
        && (el - sl > 0))
                {
                sl++;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
```

```
		|| (IsPointGoal(st, sw, sl)))
			return(front);
		else
			return(clear);
		}

	if ((ew - sw == 0)
	&& (el - sl == 0)
	&& (et - st > 0))
		{
		st++;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(up);
		else
			return(clear);
		}

	if ((ew - sw == 0)
	&& (el - sl < 0))
		{
		sl--;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(back);
		else
			return(clear);
		}

	if ((ew - sw < 0)
	&& (el - sl > 0))
		{
		sw--, sl++;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(frontleft);
		else
			return(clear);
		}

	if ((ew - sw < 0)
	&& (el - sl == 0))
		{
		sw--;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(left);
		else
			return(clear);
		}

	if ((ew - sw < 0)
	&& (el - sl < 0))
		{
		sw--, sl--;
		if ((IsPointClear(st, sw, sl))
		|| (IsPointVertex(st, sw, sl))
		|| (IsPointGoal(st, sw, sl)))
			return(backleft);
		else
			return(clear);
		}
	return(clear);
	}


float Domain::CheckLine(int st, int sw, int sl, int et, int ew, int el)
	{
//	static int displayCounter=1;
	int tmpt, tmpw, tmpl;
	float dist;

	for(dist=0.0, tmpt=et-st, tmpw=ew-sw, tmpl=el-sl;
		(tmpt != 0) || (tmpw != 0) || (tmpl != 0);
		tmpt=et-st, tmpw=ew-sw, tmpl=el-sl)
		{
		if (tmpt < 0)
			{
			return(-1.0);
```

```
                {
        if ((tmpw > 0)
        && (tmpl > 0))
                {
                dist += GetPointCost(st, sw++, sl++, frontright);
                if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
                        return(-1.0);
    continue;
                }
 if ((tmpw > 0)
        && (tmpl == 0))
                {
    dist += GetPointCost(st, sw++, sl, right);
        if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
                }
        if ((tmpw > 0)
        && (tmpl < 0))
                {
    dist += GetPointCost(st, sw++, sl--, backright);
                if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
        if ((tmpw < 0)
        && (tmpl > 0))
                {
    dist += GetPointCost(st, sw--, sl++, frontleft);
        if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
        if ((tmpw < 0)
        && (tmpl == 0))
                {
    dist += GetPointCost(st, sw--, sl, left);
        if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
        if ((tmpw < 0)
        && (tmpl < 0))
                {
    dist += GetPointCost(st, sw--, sl--, backleft);
                if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                        return(-1.0);
    continue;
                }
        if ((tmpw == 0)
        && (tmpl > 0))
                {
    dist += GetPointCost(st, sw, sl++, front);
        if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
        if ((tmpw == 0)
        && (tmpl < 0))
                {
    dist += GetPointCost(st, sw, sl--, back);
        if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
 if ((tmpw == 0)
        && (tmpl == 0)
        && (tmpt >0))
                {
    dist += GetPointCost(st++, sw, sl, up);
        if ((!IsPointClear(st, sw, sl))
        && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
    continue;
    }
        }
```

```
//      cout << "CheckLine:" << displayCounter++ << ":returned" << dist << "\n";
        return(dist);
        }


int Domain::ClearAdjPointOK(int t, int w, int l)
        {
        char type;
        int a,b;

        for(a=w-1;a<=w+1;a++)
                {
                if ((a < 0) || (a >= domainWidth))
                        continue;
                for(b=l-1;b<=l+1;b++)
                        {
                        if ((b < 0) || (b >= domainLength))
                                continue;
                        if ((a == w) && (b == l))
                                continue;

          type = GetPointType(t, a, b);
                        if ((type == OBJECT)
                        ||  (type == MOBILE_OBJECT))
                                return(FALSE);
                        }
                }
        return(TRUE);
        }


void Domain::ClearMobileObject(int t, int w, int l)
   {
        int i, j;

        if (GetPointType(t, w, l) == MOBILE_OBJECT)
                {
                if (ClearAdjPointOK(t, w, l))
                        {
                        SetPointType(t, w, l, CLEAR);
                        SetPointFrom(t, w, l, FROM_NOWHERE);
                        }
                else
                        {
                        SetPointType(t, w, l, ADJ_TO_OBJECT);
                        SetPointFrom(t, w, l, FROM_NOWHERE);
                        }
                }
        for(i=w-1, j=l-1; j<l+2;)
                {
       if ((GetPointType(t, i, j) == ADJ_TO_OBJECT)
                && (ClearAdjPointOK(t, i, j)))
                {
                SetPointType(t, i, j, CLEAR);
                        SetPointFrom(t, i, j, FROM_NOWHERE);
                        }
                if (i == w+1)
                        {
                        i = w-1;
                        j++;
                        }
                else
                        i++;
                }
        }


void Domain::ClearVerticesInTimeSlice(int t)
        {
        int w, l, pointType;

        for(w=0; w<domainWidth; w++)
                {
                for(l=0; l<domainLength; l++)
                        {
                        pointType = GetPointType(t, w, l);
                        if (pointType == VERTEX)
                                {
                                SetPointFrom(t, w, l, FROM_NOWHERE);
                                SetPointType(t, w, l, CLEAR);
                                continue;
                                }
                        if (pointType == MOBILE_OBJECT)
                                {
```

```
			ClearMobileObject(t, w, 1);
				}
			}
		}
	}

void Domain::DrawDomain()
	{
	int i;

	for (i=0; i< domainTimeSlices; i++)
		{
		DrawTimeSlice(i);
//		getch();
		}
	}

void Domain::DrawTimeSlice(int timeSlice)
	{
	char type;
	int a,b,y;

	clrscr();
	gotoxy(1,1);
	cout << "time=t" << timeSlice;

	for(b=0, y=2;b<domainLength; b++, y++)
		{
		gotoxy(1, y);
		for(a=0; a<domainWidth; a++)
			{
			type = GetPointType(timeSlice, a, b);
			switch(type)
				{
				case GOAL:
				case START:
				case ADJ_TO_OBJECT:
				case OBJECT:
				case MOBILE_OBJECT:
				case VERTEX:
					cout << type;
					break;
				case CLEAR:
					switch(GetPointFrom(timeSlice, a, b))
						{
						case FROM_RIGHT:
							cout << "R";
							break;
						case FROM_LEFT:
							cout << "L";
							break;
						case FROM_UP:
							cout << "U";
							break;
						case FROM_DOWN:
							cout << "D";
							break;
						case FROM_FRONT:
							cout << "F";
							break;
						case FROM_BACK:
							cout << "B";
							break;
						case FROM_BACKLEFT:
							cout << "T";
							break;
						case FROM_BACKRIGHT:
							cout << "U";
							break;
						case FROM_FRONTLEFT:
							cout << "V";
							break;
						case FROM_FRONTRIGHT:
							cout << "W";
							break;
						case FROM_NOWHERE:
						default:
							cout << ".";
							break;
						}
					break;
				default:
```

```
                                        cout << "?";
                                        break;
                                   }
                              }
                    cout << "\n";
                    }
          }


POINT* Domain::GetPoint(int timeSlice, int width, int length)
        {
   POINT *tmp, *tmp2;

   tmp = domainHead[timeSlice];
        tmp2 = tmp + (domainWidth*width) + length;
   return(tmp2);
   }


float Domain::GetPointCost(int timeSlice, int width, int length, int dir)
   {
   POINT *tmp;

   tmp = GetPoint(timeSlice, width, length);
        return(tmp->cost[dir]);
        }

char Domain::GetPointFrom(int timeSlice, int width, int length)
   {
   POINT *tmp;

   tmp = GetPoint(timeSlice, width, length);
        return(tmp->from);
   }


char Domain::GetPointType(int timeSlice, int width, int length)
        {
   POINT *tmp;

        tmp = GetPoint(timeSlice, width, length);
        return(tmp->type);
        }


void Domain::InitTimeSlice(int timeSlice)
        {
        POINT*tmp;
   int a,b;

        for(a=0;a<domainWidth;a++)
                {
                for(b=0;b<domainLength;b++)
                        {
          tmp = GetPoint(timeSlice, a, b);
                        tmp->dist=0.0;
                        tmp->from=FROM_NOWHERE;
                        tmp->type=CLEAR;

/*                      if (timeSlice == domainTimeSlices-1)
                                tmp->cost[FROM_UP] = BLOCKED;
                        else*/
                                tmp->cost[FROM_UP] = NORMAL;

                        if ((a == 0) && (b == 0))
                                {
                                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                                tmp->cost[FROM_RIGHT] = NORMAL;
                                tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                                tmp->cost[FROM_BACK] = BLOCKED;
                                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                                tmp->cost[FROM_LEFT] = BLOCKED;
                                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                                tmp->cost[FROM_FRONT] = NORMAL;
                                continue;
                                }
                        if ((a == 0) && (b < domainLength-1))
                                {
                                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                                tmp->cost[FROM_RIGHT] = NORMAL;
                                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                                tmp->cost[FROM_BACK] = NORMAL;
                                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                                tmp->cost[FROM_LEFT] = BLOCKED;
```

```
        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a == 0) && (b==domainLength-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = BLOCKED;
        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
        tmp->cost[FROM_FRONT] = BLOCKED;
        continue;
        }
if ((a < domainWidth-1) && (b == 0))
        {
        tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
        tmp->cost[FROM_BACK] = BLOCKED;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a < domainLength-1) && (b < domainWidth-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a < domainWidth-1) && (b == domainLength-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
        tmp->cost[FROM_RIGHT] = NORMAL;
        tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
        tmp->cost[FROM_FRONT] = BLOCKED;
        continue;
        }
if ((a == domainWidth-1) && (b == 0))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
        tmp->cost[FROM_RIGHT] = BLOCKED;
        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
        tmp->cost[FROM_BACK] = BLOCKED;
        tmp->cost[FROM_BACKLEFT] = BLOCKED;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a == domainWidth-1) && (b < domainWidth-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
        tmp->cost[FROM_RIGHT] = BLOCKED;
        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_LEFT] = NORMAL;
        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_FRONT] = NORMAL;
        continue;
        }
if ((a == domainLength-1) && (b == domainWidth-1))
        {
        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
        tmp->cost[FROM_RIGHT] = BLOCKED;
        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
        tmp->cost[FROM_BACK] = NORMAL;
        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
        tmp->cost[FROM_LEFT] = NORMAL;
```

```
                                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                                tmp->cost[FROM_FRONT] = BLOCKED;
                                continue;
                                }
                        }
                }
        }


int Domain::IsPointClear(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case CLEAR:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointGoal(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case GOAL:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }



int Domain::IsPointNearObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
                case OBJECT:
                        return(OBJECT);
                case MOBILE_OBJECT:
                        return(MOBILE_OBJECT);
                case ADJ_TO_OBJECT:
                        return(ADJ_TO_OBJECT);
                default:
                        return(FALSE);
                }
        }

int Domain::IsPointObject(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(TRUE);
        switch(GetPointType(t, w, l))
                {
```

```
			case OBJECT:
				return(OBJECT);
			case MOBILE_OBJECT:
				return(MOBILE_OBJECT);
			case ADJ_TO_OBJECT:
			default:
				return(FALSE);
			}
	}


int Domain::IsPointStart(int t, int w, int l)
	{
	if ((w < 0)
	|| (l < 0)
	|| (t < 0)
	|| (w >= domainWidth)
	|| (l >= domainLength)
	|| (t >= domainTimeSlices))
		return(FALSE);
	switch(GetPointType(t, w, l))
		{
		case START:
			return(TRUE);
		default:
			return(FALSE);
		}
	}


int Domain::IsPointVertex(int t, int w, int l)
	{
	if ((w < 0)
	|| (l < 0)
	|| (t < 0)
	|| (w >= domainWidth)
	|| (l >= domainLength)
	|| (t >= domainTimeSlices))
		return(FALSE);
	switch(GetPointType(t, w, l))
		{
		case VERTEX:
			return(TRUE);
		default:
			return(FALSE);
		}
	}


void Domain::MarkMobileObject(int t, int w, int l)
	{
	int i, j;

	if ((GetPointType(t, w, l) == CLEAR)
	|| (GetPointType(t, w, l) == ADJ_TO_OBJECT))
		{
		SetPointType(t, w, l, MOBILE_OBJECT);
		SetPointFrom(t, w, l, FROM_NOWHERE);
		}
	for(i=w-1, j=l-1; j<l+2;)
		{
		if (GetPointType(t, i, j) == CLEAR)
			{
			SetPointType(t, i, j, ADJ_TO_OBJECT);
			SetPointFrom(t, i, j, FROM_NOWHERE);
			}
		if (i == w+1)
			{
			i = w-1;
			j++;
			}
		else
			i++;
		}
	}


void Domain::MoveMobileObject(OBJECT_NODE*object)
	{
	int t=domainTimeSlices-1, w=(*object).w, l=(*object).l;

	ClearMobileObject(t, w, l);
	switch((*object).direction)
```

```
        {
        case front:
                if ((IsPointObject(t, w, l+1))
                || (IsPointStart(t, w, l+1))
                || (IsPointGoal(t, w, l+1)))
                        (*object).direction = rand() % NUM_DIRS;
                else
                        (*object).l++;
                break;
        case frontleft:
                if ((IsPointObject(t, w-1, l+1))
                || (IsPointStart(t, w-1, l+1))
                || (IsPointGoal(t, w-1, l+1)))
                        (*object).direction = rand() % NUM_DIRS;
                else
                        (*object).l++, (*object).w--;
                break;
        case left:
                if ((IsPointObject(t, w-1, l))
                || (IsPointStart(t, w-1, l))
                || (IsPointGoal(t, w-1, l)))
                        (*object).direction = rand() % NUM_DIRS;
                else
                        (*object).w--;
                break;
        case backleft:
                if ((IsPointObject(t, w-1, l-1))
                || (IsPointStart(t, w-1, l-1))
                || (IsPointGoal(t, w-1, l-1)))
                        (*object).direction = rand() % NUM_DIRS;
                else
                        (*object).w--, (*object).l--;
                break;
        case back:
                if ((IsPointObject(t, w, l-1))
                || (IsPointStart(t, w, l-1))
                || (IsPointGoal(t, w, l-1)))
                        (*object).direction = rand() % NUM_DIRS;
                else
                        (*object).l--;
                break;
        case backright:
                if ((IsPointObject(t, w+1, l-1))
                || (IsPointStart(t, w+1, l-1))
                || (IsPointGoal(t, w+1, l-1)))
                        (*object).direction = rand() % NUM_DIRS;
                else
                        (*object).w++, (*object).l--;
                break;
        case right:
                if ((IsPointObject(t, w+1, l))
                || (IsPointStart(t, w+1, l))
                || (IsPointGoal(t, w+1, l)))
                        (*object).direction = rand() % NUM_DIRS;
                else
                        (*object).w++;
                break;
        case frontright:
                if ((IsPointObject(t, w+1, l+1))
                || (IsPointStart(t, w+1, l+1))
                || (IsPointGoal(t, w+1, l+1)))
                        (*object).direction = rand() % NUM_DIRS;
                else
                        (*object).w++, (*object).l++;
                break;
        default:
                cout << "*** unknown direction for mobile object ignored ***\n";
                break;
        }
    MarkMobileObject(t, object->w, object->l);
    }


void Domain::MoveMobileObjects()
    {
    OBJECT_NODE*cur, *orig;

    orig = cur = objList.GetNextObject();
    if (cur == (OBJECT_NODE*)NULL)
    return;

    do
        {
```

```
		if ((*cur).velocity > 0)
			MoveMobileObject(cur);

	cur = objList.GetNextObject();
		} while (cur != orig);
	}


int Domain::MoveRobot(int st, int sw, int sl, int et, int ew, int el, float*dist)
	{
	int dir;

	*dist = 0.0;
	if (!IsPointStart(st, sw, sl))
		{
		*dist = 0.0;
		return(FALSE);
		}

	dir = CalcRobotDir(st, sw, sl, et, ew, el);
	SetPointType(st, sw, sl, CLEAR);
	SetPointFrom(st, sw, sl, FROM_NOWHERE);

	switch(dir)
		{
		case frontright:
			*dist = GetPointCost(st, sw, sl, frontright);
			st++, sw++, sl++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw--, sl--;
				*dist = 0.0;
				}
			break;
		case right:
			*dist = GetPointCost(st, sw, sl, right);
			st++, sw++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw--;
				*dist = 0.0;
				}
			break;
		case backright:
			*dist = GetPointCost(st, sw, sl, backright);
			st++, sw++, sl--;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw--, sl++;
				*dist = 0.0;
				}
			break;
		case frontleft:
			*dist = GetPointCost(st, sw, sl, frontleft);
			st++, sw--, sl++;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw++, sl--;
				*dist = 0.0;
				}
			break;
		case left:
			*dist = GetPointCost(st, sw, sl, left);
			st++, sw--;
			if ((!IsPointClear(st, sw, sl))
			&& (!IsPointVertex(st, sw, sl))
			&& (!IsPointGoal(st, sw, sl)))
				{
				st--, sw++;
				*dist = 0.0;
				}
			break;
		case backleft:
			*dist = GetPointCost(st, sw, sl, backleft);
			st++, sw--, sl--;
			if ((!IsPointClear(st, sw, sl))
```

```
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sw++, sl++;
                        *dist = 0.0;
                        }
                break;
        case front:
                *dist = GetPointCost(st, sw, sl, front);
                st++, sl++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sl--;
                        *dist = 0.0;
                        }
                break;
        case back:
                *dist = GetPointCost(st, sw, sl, back);
                st++, sl--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sl++;
                        *dist = 0.0;
                        }
                break;
        case up:
                *dist = GetPointCost(st, sw, sl, up);
                st++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--;
                        *dist = 0.0;
                        }
                break;
        default:        // no movement - start IS goal
                *dist = 0.0;
                break;
        }
    if (IsPointGoal(st, sw, sl))
        return(TRUE);
    else
        {
        SetPointType(st, sw, sl, START);
        SetPointFrom(st, sw, sl, FROM_NOWHERE);
        return(FALSE);
        }
    }


void Domain::SetAdjObjsInTimeSlice(int timeSlice)
    {
    int a,b;
    int w,l;

    for(w=0; w<domainWidth; w++)
        {
        for(l=0; l<domainLength; l++)
            {
            if (!IsPointObject(timeSlice, w, l))
                continue;

            for(a=w-1; a<=w+1; a++)
                {
                if ((a < 0) || (a >= domainWidth))
                    continue;
                for(b=l-1; b<=l+1; b++)
                    {
                    if ((b < 0) || (b >= domainLength))
                        continue;

                    if (GetPointType(timeSlice, a, b) == CLEAR)
                        {
                        SetPointFrom(timeSlice, a, b, FROM_NOWHERE);
                        SetPointType(timeSlice, a, b, ADJ_TO_OBJECT);
                        }
                    }
                }
            }
```

```
		}
	}


void Domain::SetGoalFromFile(char*fileName)
	{
	int i, w, l;
	char recType;
	char tmp[256];
	fstream dataFile;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Goal added.\n";

	for(;;)
		{
		dataFile. getline(tmp, sizeof(tmp));
		if (strlen(tmp) == 0)
			break;
		if (tmp[0] == GOAL)
			{
//			cout << "The GOAL line is: " << tmp << "\n";
			if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
				{
	    for(i=0; i<domainTimeSlices; i++)
	      {
				SetPointFrom(i, w, l, FROM_NOWHERE);
				SetPointType(i, w, l, GOAL);
	      }
				break;
				}
			else
				cout << "Improperly formatted line ignored\n";
			}
		}
	dataFile.close();
	}


void Domain::SetMobileObjsFromFile(char*fileName)
	{
	int w, l;
	char recType;
	char tmp[256];
	fstream dataFile;
	OBJECT_NODE* newObj;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Mobile objs added.\n";

	for(;;)
		{
		int i;

		dataFile.getline(tmp, sizeof(tmp));
		i = strlen(tmp);
		if (i <= 0)
			break;
		cout << "Length of input line is: " << i << "\n";
		if ((tmp[0] != GOAL)
		&& (tmp[0] != START))
			{
//			cout << "OBJECT line is: " << tmp << "\n";
			if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
				{
				SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
				SetPointType(domainTimeSlices-1, w, l, MOBILE_OBJECT);
				newObj = objList.BuildNewObject(w, l);
				if (newObj)
					{
					objList.InsertNewObject(newObj);
//					cout << "Added obj to obj list\n";

				}
			else
				cout << "Improperly formatted line ignored\n";
			}
		}
	dataFile.close();
	}
```

```
void Domain::SetPermObjsInTimeSlice(int timeSlice)
	{
	int w, l;

	for(w=0; w<domainWidth-5; w++)
		{
		l=2;
		switch(w)
			{
			case 3:
			case 4:
			case 5:
				break;
			default:
				SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
				SetPointType(timeSlice, w, l, OBJECT);
//				SetAdjObjsInTimeSlice(timeSlice, w, l);
				break;
			}
		}
	for(w=0; w<domainWidth-10; w++)
		{
		l=7;
		SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
		SetPointType(timeSlice, w, l, OBJECT);
//		SetAdjObjsInTimeSlice(timeSlice, w, l);
		}
	for(l=0, w=domainWidth-10; l<domainLength-2; l++)
		{
		switch(l)
			{
			case 3:
			case 4:
			case 5:
			case 6:
			case 14:
			case 15:
			case 16:
				break;
			default:
				SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
				SetPointType(timeSlice, w, l, OBJECT);
//				SetAdjObjsInTimeSlice(timeSlice, w, l);
				break;
			}
		}
	for(l=2, w=domainWidth-5; l<domainLength-7; l++)
		{
		switch(l)
			{
			case 9:
			case 10:
			case 11:
				break;
			default:
				SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
				SetPointType(timeSlice, w, l, OBJECT);
//				SetAdjObjsInTimeSlice(timeSlice, w, l);
				break;
			}
		}
	for(w=domainWidth-5, l=domainLength-7; w<domainWidth; w++)
		{
		SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
		SetPointType(timeSlice, w, l, OBJECT);
//		SetAdjObjsInTimeSlice(timeSlice, w, l);
		}
	for(w=domainWidth-10, l=domainLength-2; w<domainWidth; w++)
		{
		SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
		SetPointType(timeSlice, w, l, OBJECT);
//		SetAdjObjsInTimeSlice(timeSlice, w, l);
		}
	}


void Domain::SetPointFrom(int timeSlice, int width, int length, int from)
	{
	POINT *tmp;

	if ((width < 0)
	|| (length < 0)
	|| (timeSlice < 0)
	|| (width >= domainWidth)
```

```
	|| (length >= domainLength)
	|| (timeSlice >= domainTimeSlices))
		return;
	tmp = GetPoint(timeSlice, width, length);
	tmp->from = from;
	}

void Domain::SetPointType(int timeSlice, int width, int length, char type)
	{
	char oldType;
	POINT *tmp;

	if ((width < 0)
	|| (length < 0)
	|| (timeSlice < 0)
	|| (width >= domainWidth)
	|| (length >= domainLength)
	|| (timeSlice >= domainTimeSlices))
		return;
	tmp = GetPoint(timeSlice, width, length);
	oldType = tmp->type;
	tmp->type = type;
	if (oldType != OBJECT)
		{
//		SetAdjObjsInTimeSlice(timeSlice, width, length);
		}
	}


void Domain::SetStartFromFile(char*fileName)
	{
	int w, l;
	char recType;
	char tmp[256];
	fstream dataFile;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Start added.\n";

	for(;;)
		{
		dataFile. getline(tmp, sizeof(tmp));
		if (strlen(tmp) == 0)
			break;
		if (tmp[0] == START)
			{
//			cout << "The START line is: " << tmp << "\n";
			if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
				{
				SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
				SetPointType(domainTimeSlices-1, w, l, START);
				break;
				}
			else
				cout << "Improperly formatted line ignored\n";
			}
		}
	dataFile.close();
	}


void Domain::SetVerticesInTimeSlice(int t)
	{
	int w, l;
	int diff_counter, corner_counter;
	int side_bits, corner_bits;

	for(w=0; w<domainWidth; w++)
		{
		for(l=0; l<domainLength; l++)
			{
			if (!IsPointClear(t, w, l))
				continue;

			corner_bits = BITMASK_CLEAR;
			corner_counter = 0;
			if (IsPointNearObject(t, w-1, l-1))
				{
				corner_bits |= BITMASK_LEFT;
				corner_bits |= BITMASK_TOP;
				corner_counter++;
				}
			if (IsPointNearObject(t, w-1, l+1))
```

```
                                {
                                corner_bits |= BITMASK_LEFT;
                                corner_bits |= BITMASK_BOTTOM;
                                corner_counter++;
                                }
                        if (IsPointNearObject(t, w+1, l+1))
                                {
                                corner_bits |= BITMASK_RIGHT;
                                corner_bits |= BITMASK_BOTTOM;
                                corner_counter++;
                                }
                        if (IsPointNearObject(t, w+1, l-1))
                                {
                                corner_bits |= BITMASK_RIGHT;
                                corner_bits |= BITMASK_TOP;
                                corner_counter++;
                                }
                        if (corner_bits != BITMASK_CLEAR)
                                {
                                side_bits = BITMASK_CLEAR;
                                if (IsPointNearObject(t, w-1, l))
                                        side_bits |= BITMASK_LEFT;
                                if (IsPointNearObject(t, w, l-1))
                                        side_bits |= BITMASK_TOP;
                                if (IsPointNearObject(t, w+1, l))
                                        side_bits |= BITMASK_RIGHT;
                                if (IsPointNearObject(t, w, l+1))
                                        side_bits |= BITMASK_BOTTOM;

                                for(diff_counter=0;
                                        (side_bits != BITMASK_CLEAR)
                                        || (corner_bits != BITMASK_CLEAR);
                                        side_bits >>= 1, corner_bits >>= 1)
                                        {
                                        if ((corner_bits & (int)0x01)
                                        && ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
                                                {
                                                diff_counter++;
                                                continue;
                                                }
                                        if ((side_bits & (int)0x01)
                                        && ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
                                                {
                                                diff_counter++;
                                                continue;
                                                }
                                        }
                                if ((diff_counter > 2)
                                || ((diff_counter == 2) && (corner_counter == 1)))
                                        {
                                        SetPointFrom(t, w, l, FROM_NOWHERE);
                                        SetPointType(t, w, l, VERTEX);
                                        }
                                }
                        }
                }
        }

// edge.hpp
#define DEBUG_FILENAME "DEBUG.LOG"

typedef struct van
   {
        int t,w,l;
        float dist;
        struct van *prev, *next;
        } EDGE_NODE;

class EdgeList
        {
        public:
                EdgeList();
                ~EdgeList();
       EDGE_NODE* BuildNewEdge(int, int, int, float);
                void DelAllEdges(void);
                void DelEdge(EDGE_NODE*);
                void DelEdgeToVertex(int, int, int);
       EDGE_NODE* GetFirstEdge(void);
       EDGE_NODE* GetNextEdge(EDGE_NODE*);
       void InsertNewEdge(EDGE_NODE*);
                void ListAllEdges(int);
   private:
        EDGE_NODE *edgeHead;
        };
```

```
// edge.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include "edge.hpp"


EdgeList::EdgeList()
        {
        edgeHead = (EDGE_NODE*)NULL;
//      cout << "EdgeList Constructor\n";
        }


EdgeList::~EdgeList()
        {
        while (edgeHead != (EDGE_NODE*)NULL)
                DelEdge(edgeHead);
//      cout << "EdgeList Destructor\n";
        }


EDGE_NODE* EdgeList::BuildNewEdge(int t, int w, int l, float dist)
        {
        EDGE_NODE *a;

//      if (otherVertex != (void*)NULL)
//              {
                a = (EDGE_NODE*)farcalloc(1, sizeof(EDGE_NODE));
                if (a == (EDGE_NODE*)NULL)
                        {
                        cout << "Out of Memory in BuildNewEdge()\n";
          getch();
                        exit(0);
                        }
                (*a).t = t;
                (*a).w = w;
                (*a).l = l;
                (*a).dist = dist;
                (*a).prev = (*a).next = (EDGE_NODE*)NULL;
                return(a);
//              }
//      return((EDGE_NODE*)NULL);
        }


void EdgeList::DelAllEdges()
        {
        EDGE_NODE*a;

        for(a = GetFirstEdge();
                a != (EDGE_NODE*)NULL;
                a = GetFirstEdge())
                {
                DelEdge(a);
                }
        }

void EdgeList::DelEdge(EDGE_NODE*a)
        {
        EDGE_NODE*tmp;

//      cout << "edge 0\n";
        if (a != (EDGE_NODE*)NULL)
                {
                if ((a->prev == (EDGE_NODE*)NULL)  // del last remaining edge
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 1\n";
                        free(a);
                        edgeHead = (EDGE_NODE*)NULL;
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge in middle
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 2\n";
                        tmp = a->prev;
                        tmp->next = a->next;
```

```
                    tmp = a->next;
                    tmp->prev = a->prev;
                    free(a);
                    return;
                    }
                if ((a->prev == (EDGE_NODE*)NULL)  // del edge at sol
                && (a->next != (EDGE_NODE*)NULL))
                    {
//                  cout << "edge 3\n";
                    tmp = a->next;
                    tmp->prev = (EDGE_NODE*)NULL;
                    edgeHead = tmp;
                    free(a);
                    return;
                    }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge at eol
                && (a->next == (EDGE_NODE*)NULL))
                    {
//                  cout << "edge 4\n";
                    tmp = a->prev;
                    tmp->next = (EDGE_NODE*)NULL;
                    free(a);
                    return;
                    }
                }
        }


void EdgeList::DelEdgeToVertex(int t, int w, int l)
        {
        EDGE_NODE*tmp;

        for(tmp = GetFirstEdge(); tmp != (EDGE_NODE*)NULL; tmp = GetNextEdge(tmp))
                {
                if ((tmp->t == t)
        && (tmp->w == w)
        && (tmp->l == l))
                    {
                    DelEdge(tmp);
                    break;
                    }
                }
        }


EDGE_NODE* EdgeList::GetFirstEdge()
        {
   return(edgeHead);
   }


EDGE_NODE* EdgeList::GetNextEdge(EDGE_NODE*cur)
   {
   return(cur->next);
   }


void EdgeList::InsertNewEdge(EDGE_NODE *edge)
        {
        EDGE_NODE*cur;

        if (edge == (EDGE_NODE*)NULL)
                return;

        if (edgeHead == (EDGE_NODE*)NULL)
                {
                edgeHead = edge;
                edge->prev = edge->next = (EDGE_NODE*)NULL;
//              cout << "Inserted edge into empty list ";
//              cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//              cout << "Dist = " << edge->dist << "\n";
                return;
                }

        for(cur=edgeHead; cur != (EDGE_NODE*)NULL; cur=(*cur).next)
                {
                if ( (edge->t > cur->t)
                || ((edge->t == cur->t)
                    && (edge->w > cur->w))
                || ((edge->t == cur->t)
                    && (edge->w == cur->w)
                    && (edge->l > cur->l)) )
                    {                                       // insert after cur
                    if (cur->next == (EDGE_NODE*)NULL)
```

```
                        {
                        edge->next = (EDGE_NODE*)NULL;// no more so append to eol
                        edge->prev = cur;
                        cur->next = edge;
                        break;
                        }
                else
                        continue;                       // try next one
                }
        if ( (edge->t < cur->t)
        || ((edge->t == cur->t)
                && (edge->w < cur->w))
/*      || ((edge->t < cur->t)*/
        || ((edge->t == cur->t)
                && (edge->w == cur->w)
                && (edge->l < cur->l)) )
                {                                       // insert before cur
                if (cur->prev == (EDGE_NODE*)NULL)
                        {
                        edge->prev = (EDGE_NODE*)NULL;  // at start of list
                        edge->next = cur;
                        cur->prev = edge;
                        edgeHead = edge;
                        break;
                        }
                else
                        {
                        edge->prev = cur->prev;     // in middle/end of list
                        edge->next = cur;
                        cur->prev = edge;
                        cur = edge->prev;
                        cur->next = edge;
                        break;
                        }
                }
        if ( (edge->t == cur->t)
        && (edge->w == cur->w)
        && (edge->l == cur->l) )
                {
                /* already here - replace it ! */
/*              cout << "Already here - replacing values and disposing of new edge ";
                cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
                cout << "Dist = " << edge->dist << "\n";
                cur->w = edge->w;
                cur->l = edge->l;
                cur->dist = edge->dist;*/
//              cout << " Already here - ignoring...\n";
                cout << "!";
                free(edge);
                break;
                }
        }
//      cout << "Inserted edge into list ";
//      cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//      cout << "Dist = " << edge->dist << "\n";
        }


void EdgeList::ListAllEdges(int debugFlag)
        {
        char tmp[256];
        fstream debugFile;
        EDGE_NODE *edge;

        strcpy(tmp, "List of all edges in list\n");
        cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        for(edge = edgeHead; edge != (EDGE_NODE*)NULL; edge = (*edge).next)
                {
                sprintf(tmp, "Edge to (%d,%d,%d). Dist is %f\n",
                                        (*edge).t, (*edge).w, (*edge).l, (*edge).dist);
                cout << tmp;
                if (debugFlag)
                        {
                        debugFile.open(DEBUG_FILENAME, ios::app);
                        debugFile.write(tmp, strlen(tmp));
                        debugFile.close();
                        }
```

```
		{
	strcpy(tmp, "-----------------\n");
	cout << tmp;
	if (debugFlag)
		{
		debugFile.open(DEBUG_FILENAME, ios::app);
		debugFile.write(tmp, strlen(tmp));
		debugFile.close();
		}
	}

// object.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define NUM_DIRS 7 /* this is the 8 horizontal directions; (0->7) */
enum directions { front, back, left, right, frontright, frontleft, backright, backleft, up, down, clear};


typedef struct o
	{
	int direction, velocity;
	int w, l;
	struct o *prev, *next;
	} OBJECT_NODE;


class ObjectList
	{
	public:
		ObjectList();
		~ObjectList();
		OBJECT_NODE* BuildNewObject(int, int);
	void DelAllObjects(void);
	void DelObject(OBJECT_NODE*);
	OBJECT_NODE* GetNextObject(void);
		void InsertNewObject(OBJECT_NODE*);
	void ListAllObjects(void);
	private:
		OBJECT_NODE *objectHead;
	};

// object.cpp
#include <iostream.h>
#include <alloc.h>		// for coreleft()
#include <stdlib.h>		// for itoa()
#include <string.h>		// for strcpy()
#include <conio.h>
// #include "edge.hpp"
#include "object.hpp"

ObjectList::ObjectList()
	{
	objectHead = (OBJECT_NODE*)NULL;
	cout << "ObjectList Constructor\n";
	}



ObjectList::~ObjectList()
	{
	cout << "ObjectList Destructor\n";
	DelAllObjects();
	}



OBJECT_NODE* ObjectList::BuildNewObject(int w, int l)
	{
	OBJECT_NODE *newPtr;

	newPtr = (OBJECT_NODE*)farcalloc(1, sizeof(OBJECT_NODE));
	if (newPtr == (OBJECT_NODE*)NULL)
		{
		cout << "Out of memory in BuildNewObject()\n";
	getch();
```

```
                return(NULL);
                }
        (*newPtr).direction = rand() % NUM_DIRS;
        (*newPtr).velocity = 1;
        (*newPtr).prev = (*newPtr).next = (OBJECT_NODE*)NULL;
        (*newPtr).w = w;
        (*newPtr).l = l;
        return(newPtr);
        }


void ObjectList::DelAllObjects()
        {
        OBJECT_NODE *tmp;

        while (objectHead != (OBJECT_NODE*)NULL)
                {
                tmp = objectHead;
                objectHead = (*objectHead).next;
                DelObject(tmp);
                }
        }


void ObjectList::DelObject(OBJECT_NODE *todie)
   {
        OBJECT_NODE *cur;

        if (((*todie).prev != (OBJECT_NODE*)NULL)
        && ((*todie).next != (OBJECT_NODE*)NULL))
                {
//              cout << "Deleted Object (" << todie->w << "," << todie->l << ")\n";
                cur=(*todie).prev;
                if (cur == todie)
                        {
                        objectHead = (OBJECT_NODE*)NULL;
//                      cout << "Object List Empty\n";
                        free(todie);
                        return;
                        }
                else
                        {
                        (*cur).next=(*todie).next;
                        cur = (*todie).next;
                        (*cur).prev = (*todie).prev;
                        if (objectHead == todie)
                                objectHead = (*todie).next;
        free(todie);
          return;
                        }
                }
        cout << "***Did NOT delete rotton Object (" << todie->w << "," << todie->l << ")\n";
        }


OBJECT_NODE* ObjectList::GetNextObject()
   {
   OBJECT_NODE*tmp;

        if (objectHead == (OBJECT_NODE*)NULL)
                return((OBJECT_NODE*)NULL);

        tmp = objectHead;
   objectHead = objectHead->next;
   return(tmp);
   }


void ObjectList::InsertNewObject(OBJECT_NODE*newPtr)
        {
        OBJECT_NODE *cur;

        if (newPtr == (OBJECT_NODE*)NULL)
                return;

        if (objectHead == (OBJECT_NODE*)NULL)
                {
                objectHead = newPtr;
                (*newPtr).prev = (*newPtr).next = newPtr;
                }
        else
                {
```

```
                /* insert before first node */
                cur = objectHead;
                (*newPtr).next = cur;
                (*newPtr).prev = (*cur).prev;
                (*cur).prev = newPtr;

                cur = (*newPtr).prev;
                (*cur).next = newPtr;
                objectHead = newPtr;
           }
//      cout << "Inserted object (" << newPtr->w << "," << newPtr->l << ")\n";
        }


void ObjectList::ListAllObjects()
        {
   OBJECT_NODE*tmp;

   if (objectHead == (OBJECT_NODE*)NULL)
      {
      cout << "ObjectList is empty\n";
      return;
      }
   for(tmp=objectHead;;tmp = tmp->next)
      {
       cout << "Object:(" << tmp->w << "," << tmp->l << ")\n";
      tmp = tmp->next;
      if (tmp == objectHead)
         {
         cout << "ObjectList ended\n";
         break;
         }
      }
       }

// vertex.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00

#define DEBUG_FILENAME "DEBUG.LOG"

typedef struct vn
        {
        int t,w,l;
        char nodeType;
        struct vn *searchPrev, *searchNext;
        float searchDist;
        int searchMarker;
        EdgeList *edgeList;
        struct vn *pathFrom, *pathTo;
        struct vn *prev, *next;
        } VERTEX_NODE;


class VertexList
        {
        public:
                VertexList();
                ~VertexList();
                void AddToSearchList(VERTEX_NODE*);
        VERTEX_NODE* BuildNewVertex(int,int,int,char);
                int CalcRobotDir(VERTEX_NODE*);
                void DelAllVertices(void);
        void DelVertex(VERTEX_NODE*);
        int FindPath(void);
        VERTEX_NODE* FindVertex(int, int, int);
                VERTEX_NODE* GetFirstVertex(void);
                VERTEX_NODE* GetGoalVertex(void);
                VERTEX_NODE* GetNextVertex(VERTEX_NODE*);
```

```
        int GetSearchMarker(void);
    VERTEX_NODE* GetStartVertex(void);
    void InsertAllVertices(void);
        void InsertNewVertex(VERTEX_NODE*);
        void ListAllVertices(int);
    void ListSearchList(int);
//      void MoveRobot(int, int, int, int);
        void MarkPath(VERTEX_NODE*);
        void RemoveFromSearchList(VERTEX_NODE*);
    void RemoveVertex(VERTEX_NODE*);
        void TrimSearchList(void);
    private:
        VERTEX_NODE* vertexHead;
    VERTEX_NODE* searchHead;
    int searchMarker;
    float searchTrimDist;
    };

// vertex.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include "edge.hpp"
#include "vertex.hpp"

VertexList::VertexList()
    {
    vertexHead = (VERTEX_NODE*)NULL;
  searchHead = (VERTEX_NODE*)NULL;
  searchMarker = 0;
  searchTrimDist = -1.0;
  cout << "Initialised Vertex class\n";
    }



VertexList::~VertexList()
    {
    cout << "VertexList destructor start-:\n";
    DelAllVertices();
    cout << "VertexList destructor ended!\n";
    }


void VertexList::AddToSearchList(VERTEX_NODE*a)
    {
    VERTEX_NODE *tmp;

    if (a == (VERTEX_NODE*)NULL)
        return;

    if (((*a).searchPrev != (VERTEX_NODE*)NULL) // if already in fringe
    || ((*a).searchNext != (VERTEX_NODE*)NULL)) // list, dont add again
        return;

    if (searchHead == (VERTEX_NODE*)NULL)
        {
        searchHead = a;
        a->pathFrom = a->pathTo = (VERTEX_NODE*)NULL;
        a->searchPrev = a->searchNext = a;
        return;
        }

  for (tmp = searchHead;tmp != (VERTEX_NODE*)NULL;)
    {
        if (a->searchDist <= tmp->searchDist)
            {
            a->searchPrev = tmp->searchPrev;
            a->searchNext = tmp;
            tmp->searchPrev = a;
            tmp = a->searchPrev;
            tmp->searchNext = a;
            if (searchHead == a->searchNext)
                searchHead = a;
            return;
            }
        if (tmp->searchNext == searchHead)
            {
            a->searchPrev = tmp;
            a->searchNext = searchHead;
```

```
                        tmp->searchNext = searchHead; searchPrev = a;
                        return;
                        }
                else
                        tmp = tmp->searchNext;
                }
        }


VERTEX_NODE* VertexList::BuildNewVertex(int t, int w, int l, char nodeType)
        {
        VERTEX_NODE *newPtr;

        newPtr = (VERTEX_NODE*)farcalloc(1, sizeof(VERTEX_NODE));
        if (newPtr == (VERTEX_NODE*)NULL)
                {
                cout << "Farcalloc() failed in BuildNewVertex!\n";
        getch();
                exit(0);
                }
        (*newPtr).t = t;
        (*newPtr).w = w;
        (*newPtr).l = l;
        (*newPtr).nodeType = nodeType;
        (*newPtr).edgeList = new EdgeList();
        (*newPtr).pathTo = (*newPtr).pathFrom = (VERTEX_NODE*)NULL;
        (*newPtr).searchDist = 0.0;
        (*newPtr).searchPrev = (*newPtr).searchNext = (VERTEX_NODE*)NULL;
        (*newPtr).prev = (*newPtr).next = (VERTEX_NODE*)NULL;
        if (nodeType == START)
                {
                (*newPtr).searchMarker = searchMarker+1;
                AddToSearchList(newPtr);
                }
        else
                (*newPtr).searchMarker = searchMarker;
        return(newPtr);
        }


void VertexList::DelAllVertices()
   {
        while (vertexHead != (VERTEX_NODE*)NULL)
                {
//              cout << "About to delete (" << vertexHead->t << "," << vertexHead->w ;
//              cout << "," << vertexHead->l << ") at addr: " << vertexHead ;
//              cout << ": Prev:" << vertexHead->prev << ": Next:" << vertexHead->next <<"\n";
                DelVertex(vertexHead);
                }
        cout << "----------\n";
//      getch();
        }


void VertexList::DelVertex(VERTEX_NODE *todie)
        {
        VERTEX_NODE *tmp;
        EDGE_NODE*a;

        if (todie == (VERTEX_NODE*)NULL)
                return;

        for(a = todie->edgeList->GetFirstEdge();
                a != (EDGE_NODE*)NULL;
                a = todie->edgeList->GetFirstEdge())
                {
        todie->edgeList->DelEdgeToVertex(a->t, a->w, a->l);
        tmp = FindVertex(a->t, a->w, a->l);
        if (tmp != (VERTEX_NODE*)NULL)
           {
           tmp->edgeList->DelEdgeToVertex(todie->t, todie->w, todie->l);
           }
//              tmp = (VERTEX_NODE*)(a->otherVertex);
//              cout << "On vertex (" << todie->t << "," << todie->w << ",";
//              cout << todie->l << "):Del edge to (" << tmp->t << "," << tmp->w << ",";
//              cout << tmp->l << ")\n";
//              cout << "todie at:" << todie << ". a at:" << a << "\n";
//              cout << "  tmp at:" << tmp << "\n";
//              todie->edgeList->DelEdgeToVertex((void*)tmp);
//              cout << "On vertex (" << tmp->t << "," << tmp->w << ",";
//              cout << tmp->l << "):Del edge to (" << todie->t << "," << todie->w << ",";
//              cout << todie->l << ")\n";
//              tmp->edgeList->DelEdgeToVertex((void*)todie);
```

```
		}
	delete todie->edgeList;
	RemoveFromSearchList(todie);

	if ( ((*todie).prev == (VERTEX_NODE*)NULL)
	&& ((*todie).next == (VERTEX_NODE*)NULL) )
		{
		vertexHead = (VERTEX_NODE*)NULL;
		free(todie);
		return;
		}
	if ( ((*todie).prev != (VERTEX_NODE*)NULL)
	&& ((*todie).next != (VERTEX_NODE*)NULL) )
		{
		tmp = (*todie).prev;
		(*tmp).next = (*todie).next;
		tmp = (*todie).next;
		(*tmp).prev = (*todie).prev;
		free(todie);
		return;
		}
	if ( ((*todie).prev == (VERTEX_NODE*)NULL)
	&& ((*todie).next != (VERTEX_NODE*)NULL) )
		{
		vertexHead = tmp = (*todie).next;
		(*tmp).prev = (VERTEX_NODE*)NULL;
		free(todie);
		return;
		}
	if ( ((*todie).prev != (VERTEX_NODE*)NULL)
	&& ((*todie).next == (VERTEX_NODE*)NULL) )
		{
		tmp = (*todie).prev;
		(*tmp).next = (VERTEX_NODE*)NULL;
		free(todie);
		return;
		}
	}


int VertexList::FindPath()
	{
	EDGE_NODE *e;
	int goalFound=FALSE;
	float dist;
	VERTEX_NODE *cur, *adj, *dest, *t1, *t2;

	if (searchHead->nodeType == START)
		searchMarker = searchHead->searchMarker;
	else
		return(FALSE);

	t1 = GetStartVertex();
	t2 = GetGoalVertex();
	if ((t1 != (VERTEX_NODE*)NULL)
	&& (t2 != (VERTEX_NODE*)NULL)
	&& (t1->w == t2->w)
	&& (t1->l == t2->l))
		{
		t1->pathTo = t2;
		return(TRUE);
		}

	for (cur = searchHead; cur != (VERTEX_NODE*)NULL; cur = searchHead)
		{
		if ((cur->searchDist >= searchTrimDist)
		&& (searchTrimDist > 0.0))
			{
			RemoveFromSearchList(cur);
			continue;
			}
		for(e = cur->edgeList->GetFirstEdge();
			e != (EDGE_NODE*)NULL;
			e = cur->edgeList->GetNextEdge(e))
			{
	  adj = FindVertex(e->t, e->w, e->l);
	  if (adj == (VERTEX_NODE*)NULL)
	     continue;

//	   adj = (VERTEX_NODE*) e->otherVertex;
			dist = cur->searchDist + e->dist;
			if ( (adj->searchMarker != searchMarker)
			|| ((adj->searchMarker == searchMarker)
				&& (adj->searchDist > dist)) )
```

```
                                {
                                if ((dist <= searchTrimDist)
                                || (searchTrimDist <= 0.0))
                                        {
                                        adj->pathFrom = cur;
                                        adj->searchMarker = searchMarker;
                                        adj->searchDist = dist;
                                        AddToSearchList(adj);
                                        if (adj->nodeType == GOAL)
                                                {
                                                goalFound = TRUE;
                                                searchTrimDist = dist;
                                                MarkPath(adj);
                                                TrimSearchList();
                                                }
                                        }
                                }
                        }
                RemoveFromSearchList(cur);
                }
        return(goalFound);
        }


VERTEX_NODE* VertexList::FindVertex(int t, int w, int l)
    {
    VERTEX_NODE* cur;

    for(cur = GetFirstVertex();
        cur != (VERTEX_NODE*)NULL;
        cur = GetNextVertex(cur))
        {
        if ((cur->t == t)
        && (cur->w == w)
        && (cur->l == l))
            return(cur);            // found it

        if ((cur->t >= t)
        && (cur->w >= w)
        && (cur->l >= l))
            break;                  // passed it - it's not in the list
        }
    return((VERTEX_NODE*)NULL);
    }


VERTEX_NODE* VertexList::GetFirstVertex()
    {
    return(vertexHead);
    }


VERTEX_NODE* VertexList::GetGoalVertex()
    {
    VERTEX_NODE*tmp;

    for(tmp = GetFirstVertex();
        tmp != (VERTEX_NODE*)NULL;
        tmp = GetNextVertex(tmp))
        {
                if (tmp->nodeType == GOAL)
            break;
        }
    return(tmp);
    }


VERTEX_NODE* VertexList::GetNextVertex(VERTEX_NODE*cur)
    {
    return(cur->next);
    }


int VertexList::GetSearchMarker()
        {
        return(searchMarker);
        }


VERTEX_NODE* VertexList::GetStartVertex()
        {
        VERTEX_NODE*tmp;
```

```
	for(tmp = GetFirstVertex();
		tmp != (VERTEX_NODE*)NULL;
		tmp = GetNextVertex(tmp))
		{
		if (tmp->nodeType == START)
	  break;
	}
    return(tmp);
    }


void VertexList::InsertNewVertex(VERTEX_NODE*a)
	{
	VERTEX_NODE *cur;

	if (a == (VERTEX_NODE*)NULL)
		return;

	if (vertexHead == (VERTEX_NODE*)NULL)
		{
		vertexHead = a;
		(*a).prev = (*a).next = (VERTEX_NODE*)NULL;
		return;
		}

	for(cur=vertexHead; cur != (VERTEX_NODE*)NULL;   cur=(*cur).next)
		{
		if ( (a->t > cur->t)
	|| ((a->t == cur->t)
	  && (a->w > cur->w))
	 || ((a->t == cur->t)
	  && (a->w == cur->w)
		&& (a->l > cur->l)) )
			{				// insert after cur
			if ((*cur).next == (VERTEX_NODE*)NULL)
				{
				(*a).next = (*cur).next;     // eol - append new node
				(*a).prev = cur;
				(*cur).next = a;
				break;
				}
			else
				continue;			// get next node
			}
		if ( (a->t < cur->t)
		|| ((a->t == cur->t)
			&& (a->w < cur->w))
		|| ((a->t == cur->t)
			&& (a->w == cur->w)
			&& (a->l < cur->l)) )
			{				 // insert before cur
			if (cur->prev == (VERTEX_NODE*)NULL)
				{
				a->prev = (VERTEX_NODE*)NULL;// at start of list
				a->next = cur;
				cur->prev = a;
				vertexHead = a;
				break;
				}
			else
				{
				a->prev = cur->prev;          // in middle/end of list
				a->next = cur;
				cur->prev = a;
				cur = a->prev;
				cur->next = a;
				break;
				}
			}
		if ((a->t == cur->t)
		&& (a->w == cur->w)
		&& (a->l == cur->l))
			{				 // insert after cur at eol
			if ((*cur).next == (VERTEX_NODE*)NULL)
				{
				(*a).next = (VERTEX_NODE*)NULL;
				(*a).prev = cur;
				(*cur).next = a;
				break;
				}
			else
				continue;
			}
		}
```

```
//      AddToSearchList(a);
//      cout << "Inserted vertex (";
// cout << (*a).t << "," << (*a).w << "," << (*a).l << ")\n";
        }


void VertexList::ListAllVertices(int debugFlag)
        {
        char tmp[256];
        fstream debugFile;
        VERTEX_NODE *cur;

        strcpy(tmp, "List of all nodes in list\n");
        cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        for(cur=vertexHead; cur != (VERTEX_NODE*)NULL; cur=cur->next)
                {
                sprintf(tmp, "Vertex:%c: at (%d,%d,%d)\n",
                                (*cur).nodeType, (*cur).t, (*cur).w, (*cur).l);
                cout << tmp;
                if (debugFlag)
                        {
                        debugFile.open(DEBUG_FILENAME, ios::app);
                        debugFile.write(tmp, strlen(tmp));
                        debugFile.close();
                        }
                cur->edgeList->ListAllEdges(debugFlag);
                }
        }


void VertexList::ListSearchList(int debugFlag)
        {
        char tmp[256];
        fstream debugFile;
        VERTEX_NODE*cur;

        if (searchHead == (VERTEX_NODE*)NULL)
                {
                sprintf(tmp,"Empty SearchList! (marker=%d)\n", searchMarker);
      cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
                return;
                }

        sprintf(tmp, "List of nodes in Search list (marker=%d)\n", searchMarker);
   cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        cur = searchHead;
        do
                {
                sprintf(tmp, "SearchList node (%d,%d,%d) Dist = %f",
              cur->t, cur->w, cur->l, cur->searchDist);
      cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
                cur=cur->searchNext;
                }
        while (cur != searchHead);
   strcpy(tmp, "End of SearchList\n");
   cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
```

```
                debugFile.close();
                }
        }


void VertexList::MarkPath(VERTEX_NODE*v)
        {
        VERTEX_NODE*tmp;

        if (v == (VERTEX_NODE*)NULL)
                return;

//      cout << "Path from Goal to Start\n";
        v->pathTo = (VERTEX_NODE*)NULL;
        do
                {
                tmp = v->pathFrom;
//              cout << "(" << v->t << "," << v->w << "," << v->l << ")\n";
                if (tmp != (VERTEX_NODE*)NULL)
                        {
                        tmp->pathTo = v;
                        v = v->pathFrom;
                        }
                }
        while (tmp != (VERTEX_NODE*)NULL);
        }


void VertexList::RemoveFromSearchList(VERTEX_NODE*a)
        {
        VERTEX_NODE *cur;

        if ((a->searchPrev == (VERTEX_NODE*)NULL)
        || (a->searchNext == (VERTEX_NODE*)NULL))
        {
        if (searchHead == a)
           searchHead = (VERTEX_NODE*)NULL;
        return;
        }

        cur=a->searchPrev;
        if (cur == a)
                {
        searchHead = (VERTEX_NODE*)NULL;
//              cout << "Removed Search Node (";
//      cout << a->t << "," << a->w << "," << a->l << "). SearchList empty\n";
                }
        else
                {
                cur->searchNext = a->searchNext;
                cur = a->searchNext;
                cur->searchPrev = a->searchPrev;
                if (searchHead == a)
                        searchHead = a->searchNext;
//              cout << "Removed Search Node (";
//      cout << a->t << "," << a->w << "," << a->l << ").\n";
                }
        a->searchPrev = a->searchNext = (VERTEX_NODE*)NULL;
        }


void VertexList::RemoveVertex(VERTEX_NODE *todie)
        {
        VERTEX_NODE *tmp;

        if (todie == (VERTEX_NODE*)NULL)
                return;

        RemoveFromSearchList(todie);

        if ( ((*todie).prev == (VERTEX_NODE*)NULL)
        && ((*todie).next == (VERTEX_NODE*)NULL) )
                {
                vertexHead = (VERTEX_NODE*)NULL;
                return;
                }
        if ( ((*todie).prev != (VERTEX_NODE*)NULL)
        && ((*todie).next != (VERTEX_NODE*)NULL) )
                {
                tmp = (*todie).prev;
                (*tmp).next = (*todie).next;
                tmp = (*todie).next;
                (*tmp).prev = (*todie).prev;
                return;
```

```
        }
    if ( ((*todie).prev == (VERTEX_NODE*)NULL)
    && ((*todie).next != (VERTEX_NODE*)NULL) )
            {
            vertexHead = tmp = (*todie).next;
            (*tmp).prev = (VERTEX_NODE*)NULL;
            return;
            }
    if ( ((*todie).prev != (VERTEX_NODE*)NULL)
    && ((*todie).next == (VERTEX_NODE*)NULL) )
            {
            tmp = (*todie).prev;
            (*tmp).next = (VERTEX_NODE*)NULL;
            return;
            }
    }

void VertexList::TrimSearchList()
   {
       VERTEX_NODE *cur, *tmpPtr=(VERTEX_NODE*)0;
       int trimmed=TRUE;

//     cout << "Trimming search list to " << searchTrimDist << " or less\n";
       if ((searchHead == (VERTEX_NODE*)NULL)
   || (searchTrimDist < 0.0))
              return;

   cur = searchHead;
       while ((tmpPtr != cur) || (trimmed == TRUE))
              {
              if (trimmed==TRUE)
                     {
                     tmpPtr=cur;
                     trimmed=FALSE;
                     }
              if ((*cur).searchDist >= searchTrimDist)
                     {
//                   cout << "Trimmed out (";
//          cout << cur->t << "," << cur->w << "," << cur->l << ") ";
//          cout << "Dist was " << cur->searchDist << "\n";
                     if (cur == cur->searchNext)
                            {
          RemoveFromSearchList(cur);
                            break;
                            }
                     else
                            {
                            tmpPtr = cur->searchNext;
                            RemoveFromSearchList(cur);
                            cur = tmpPtr;
                            trimmed = TRUE;
                            }
                     }
              else
                     cur = cur->searchNext;
              }
//   cout << "-------\n";
   }
```

B.10. Partial Update A* Graph Theory

The Partial Update A* Search Algorithm for Graph Theory which was developed and implemented as part of this project was coded in ANSI C++. The filenames for each of the separate source code files were supplied inside C++ format comments (i.e. //) at the beginning of each file listing. A detailed explanation of the design behind this program was presented in Chapter 4.

```
// main.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>          // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <sys\timeb.h>
```

```
#include <dir.h>
#include <ctype.h>
#include "edge.hpp"
#include "vertex.hpp"
#include "object.hpp"
#include "domain.hpp"
#include "bench.hpp"

#define MAPFILE_MASK "MAP*.DAT"

int main(int, char**);
void BuildUpdateListFromDomain(VertexList&, VertexList&, Domain&);
void BuildVertexListFromDomain(VertexList&, Domain&);
void FindAllEdges(VertexList&, Domain&);
void FindEdgesForUpdateNode(VERTEX_NODE*, VertexList&, VertexList&, Domain&);
int MemStatus(char*);
void MergeLists(VertexList&, VertexList&, Domain&);
int SetWorkingDir(void);
void RenameMapFile(char*);


int main(int argc, char** argv)
	{
	char* shortName;
	struct ffblk mapfile;

	for(;;)
		{
		shortName = strstr(argv[0], "\\");
		if (shortName == (char*)NULL)
			{
			shortName = argv[0];
			cout << "This is " << shortName << "\n";
			break;
			}
		else
			argv[0] = shortName+1;
		}

	MemStatus("Free memory before mainloop in main: ");

	if (!SetWorkingDir())
		{
		cout << "Program exiting gracefully\n";
		return(1);
		}
	if (findfirst(MAPFILE_MASK, &mapfile, 0))
		{
		cout << "Program exiting gracefully - no map files found\n";
		return(1);
		}
	do
		{
		char tmp[256];
		int i;
		Benchmark stopWatch(shortName);

		sprintf(tmp, "Starting on %s: ", mapfile.ff_name);
		MemStatus(tmp);
		for (i=0; i<NUMTIMES; i++)
			{
			int timeTaken;
			float distTravelled;
			Domain theWorld(10, 20, 20, mapfile.ff_name);
			VertexList vertList, updateList;

			timeTaken = 0;
			distTravelled = 0.0;
			stopWatch.IterStart(i, mapfile.ff_name);
			stopWatch.Click();
			BuildVertexListFromDomain(vertList, theWorld);
			FindAllEdges(vertList, theWorld);
			for(;;)
				{
				int pathFoundFlag, goalFoundFlag, a, b, c;
				float dist;
				VERTEX_NODE *t1;

				stopWatch.Click();
				theWorld.DrawDomain();
				stopWatch.Click();
				pathFoundFlag = vertList.FindPath();
				stopWatch.Click();
```

```
                                if (pathFoundFlag == TRUE)
                                        {
                                        cout << "Path found to the GOAL!";
                                        cout << "(dist=" << distTravelled << ")\n";
                                        }
                                else
                                        {
                                        cout << "No path found to the GOAL!";
                                        cout << "(dist=" << distTravelled << ")\n";
                                        }

                                t1 = vertList.GetStartVertex();
                                if (pathFoundFlag)
                                        {
                                        a = t1->pathTo->t;
                                        b = t1->pathTo->w;
                                        c = t1->pathTo->l;
                                        }
                                else
                                        {
                                        a = (t1->t)+1;
                                        b = t1->w;
                                        c = t1->l;
                                        }
                                goalFoundFlag = theWorld.MoveRobot(t1->t, t1->w, t1->l,
a, b, c, &dist);
                                distTravelled += dist;
                                timeTaken++;
                                if (goalFoundFlag)
                                        {
                                        cout << "MADE IT TO THE GOAL!\n";
                                        break;
                                        }

                                theWorld.AdvanceTime();
                                stopWatch.Click();
                                BuildUpdateListFromDomain(vertList, updateList, theWorld);
                                MergeLists(vertList, updateList, theWorld);
                                }
                        stopWatch.IterStop(timeTaken, distTravelled);
                        }
                stopWatch.LogCalcs();
                RenameMapFile(mapfile.ff_name);
                }
        while (!findnext(&mapfile));

        MemStatus("Free memory after mainloop in main: ");
        return(0);
        }


void BuildUpdateListFromDomain(VertexList &vList, VertexList &uList, Domain &domain)
        {
        char type;
        int t=domain.GetDomainTimeSlices(), a;
        int w=domain.GetDomainWidth(), b;
        int l=domain.GetDomainLength(), c;
        VERTEX_NODE*newPtr;

        for(a=0; a<t; a++)
          {
                for(b=0; b<w; b++)
                        {
                        for(c=0; c<l; c++)
                                {
                                type = domain.GetPointType(a, b, c);
                                if ((type == VERTEX)
                                || (type == START)
                                || (type == GOAL))
                                        {
                                        VERTEX_NODE *tmpv, *tmpu;

                                        tmpv = vList.FindVertex(a, b, c);
                                        tmpu = uList.FindVertex(a, b, c);

                                        if ((!tmpv)
                                        && (!tmpu))
                                                {
                                                newPtr = uList.BuildNewVertex(a, b, c, type);
                                                uList.InsertNewVertex(newPtr);
                                                continue;
                                                }
```

```
                        if ((tmpv != (VERTEX_NODE*)NULL)
                        && (tmpv->nodeType != type))
                                {
                                vList.RemoveVertex(tmpv);
                                vList.DelVertex(tmpv);
                                newPtr = uList.BuildNewVertex(a, b, c, type);
                                uList.InsertNewVertex(newPtr);
                                continue;
                                }

                        if ((tmpu != (VERTEX_NODE*)NULL)
                        && (tmpu->nodeType != type))
                                {
                                uList.RemoveVertex(tmpu);
                                uList.DelVertex(tmpu);
                                newPtr = uList.BuildNewVertex(a, b, c, type);
                                uList.InsertNewVertex(newPtr);
                                continue;
                                }
                        }
                else
                        {
                        VERTEX_NODE *v;

                        v = vList.FindVertex(a, b, c);
                        if (v!=(VERTEX_NODE*)NULL)
                                {
                                EDGE_NODE*e;

                                for(e = v->edgeList->GetFirstEdge();
                                        e != (EDGE_NODE*)NULL;
                                        e = v->edgeList->GetNextEdge(e))
                                        {
                                        VERTEX_NODE* v2;

                                        v2 = vList.FindVertex(e->t, e->w, e->l);
                                        if (v2 != (VERTEX_NODE*)NULL)
                                                {
                                                vList.RemoveVertex(v2);
                                                uList.InsertNewVertex(v2);
                                                v2->edgeList->DelAllEdges();
                                                }
                                        }
                                vList.DelVertex(v);
                                }
                        v = uList.FindVertex(a, b, c);
                        if (v != (VERTEX_NODE*)NULL)
                                {
                                uList.DelVertex(v);
                                }
                        }
                }
        }
}


void BuildVertexListFromDomain(VertexList &vList, Domain &domain)
  {
        char type;
        int t=domain.GetDomainTimeSlices(), a;
        int w=domain.GetDomainWidth(), b;
        int l=domain.GetDomainLength(), c;
        VERTEX_NODE*newPtr, *gPtr, *sPtr;

        for(gPtr = sPtr = (VERTEX_NODE*)NULL, a=0; a<t; a++)
          {
                for(b=0; b<w; b++)
                        {
                        for(c=0; c<l; c++)
                                {
                                type = domain.GetPointType(a, b, c);
                                if ((type == VERTEX)
                                || (type == START)
                                || (type == GOAL))
                                        {
                                newPtr = vList.BuildNewVertex(a, b, c, type);
                                vList.InsertNewVertex(newPtr);
                                        }
              if (type == GOAL)
                 gPtr = newPtr;
              if (type == START)
                 sPtr = newPtr;
```

```
            }
    if ((gPtr != (VERTEX_NODE*)NULL)
    && (sPtr != (VERTEX_NODE*)NULL))
        {
        sPtr->searchDist = 0.0;
                sPtr->estDist = vList.CalcEstimatedDist(sPtr->w, sPtr->l, gPtr->w, gPtr->l);
                vList.AddToSearchList(sPtr);
        }
        }


void FindAllEdges(VertexList &vList, Domain &domain)
        {
        EDGE_NODE*e;
        VERTEX_NODE*a,*b;
        float dist;

        for(a = vList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = vList.GetNextVertex(a))
                {
                for(b = vList.GetNextVertex(a);
                        b != (VERTEX_NODE*)NULL;
                        b = vList.GetNextVertex(b))
          {
                        if (a == b)
                                continue;

                        if ((a->t == b->t)
                        && (a->w == b->w)
                        && (a->l == b->l))
                                continue;

          dist = domain.CheckLine(a->t, a->w, a->l, b->t, b->w, b->l);
          if (dist > 0.0)
              {
                                e = a->edgeList->BuildNewEdge(b->t, b->w, b->l, dist);
                                a->edgeList->InsertNewEdge(e);
                                }
                        dist = domain.CheckLine(b->t, b->w, b->l, a->t, a->w, a->l);
                        if (dist > 0.0)
                                {
                                e = b->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                                b->edgeList->InsertNewEdge(e);
                                }
                        }
                }
        }

void FindEdgesForUpdateNode(VERTEX_NODE*cur, VertexList &vList, VertexList &uList, Domain &domain)
        {
        EDGE_NODE*e;
        VERTEX_NODE*a;
        float dist;

        for(a = vList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = vList.GetNextVertex(a))
                {
      if (a == cur)
                continue;

                if ((a->t == cur->t)
                && (a->w == cur->w)
                && (a->l == cur->l))
                        continue;

      dist = domain.CheckLine(a->t, a->w, a->l, cur->t, cur->w, cur->l);
      if (dist > 0.0)
          {
                        e = a->edgeList->BuildNewEdge(cur->t, cur->w, cur->l, dist);
                        a->edgeList->InsertNewEdge(e);
                        }
                dist = domain.CheckLine(cur->t, cur->w, cur->l, a->t, a->w, a->l);
                if (dist > 0.0)
                        {
                        e = cur->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                        cur->edgeList->InsertNewEdge(e);
                        }
                }
```

```
        for(a = uList.GetFirstVertex();
                a != (VERTEX_NODE*)NULL;
                a = uList.GetNextVertex(a))
                {
        if (a == cur)
                continue;

                if ((a->t == cur->t)
                && (a->w == cur->w)
                && (a->l == cur->l))
                        continue;

        dist = domain.CheckLine(a->t, a->w, a->l, cur->t, cur->w, cur->l);
        if (dist > 0.0)
                        {
                        e = a->edgeList->BuildNewEdge(cur->t, cur->w, cur->l, dist);
                        a->edgeList->InsertNewEdge(e);
                        }
                dist = domain.CheckLine(cur->t, cur->w, cur->l, a->t, a->w, a->l);
                if (dist > 0.0)
                        {
                        e = cur->edgeList->BuildNewEdge(a->t, a->w, a->l, dist);
                        cur->edgeList->InsertNewEdge(e);
           }
                }
        }


int MemStatus(char *StatusMessage)
        {
        char tmp[256];
        fstream debugFile;
        long MemLeft;
        int ret;

        debugFile.open("DEBUG.LOG", ios::app);
        if (!debugFile)
                cout << "Unable to open DEBUG.LOG\n";

        MemLeft = (long) coreleft();
        sprintf(tmp, "%s%ld\n", StatusMessage, MemLeft);
        debugFile.write(tmp, strlen(tmp));
        debugFile.close();
//      cout << StatusMessage << MemLeft << "\n";

        ret = farheapcheck();
        if (ret == _HEAPOK)
                cout << "Heap ok" << StatusMessage << "\n";
        else
                {
                cout << "Heap error <" << ret << ">" << StatusMessage<< "\n";
                getch();
                return(FALSE);
                }
        return(TRUE);
        }


void MergeLists(VertexList &vList, VertexList &uList, Domain &domain)
   {
   VERTEX_NODE* cur;

   for (cur=uList.GetFirstVertex();
      cur!=(VERTEX_NODE*)NULL;
      cur=uList.GetFirstVertex())
      {
      uList.RemoveVertex(cur);
                FindEdgesForUpdateNode(cur, vList, uList, domain);
                vList.InsertNewVertex(cur);
                }
       cur = vList.GetStartVertex();
       if (cur != (VERTEX_NODE*)NULL)
                {
                cur->searchMarker = vList.GetSearchMarker() + 1;
                vList.AddToSearchList(cur);
                }
        }


int SetWorkingDir()
        {
        char mapdir[256];
```

```
	cout <<"Enter the directory containing map files, or \"q\" for quit:";
	cin >> mapdir;
	if (toupper(mapdir[0]) == 'Q')
		{
		cout << "Quitting...\n";
		return(FALSE);
		}
	if (chdir(mapdir))
		{
		cout << "The directory " << mapdir << " could not be found.\n";
		return(FALSE);
		}
	cout << "Made " << mapdir << " the current directory.\n";
	return(TRUE);
	}


void RenameMapFile(char*filename)
	{
	char newfilename[128];
	char* ch;
	int i=(int)'.';

	strcpy(newfilename, filename);
	ch = strrchr(newfilename,i);
	if (ch != (char*)NULL)
		{
		strcpy(ch, ".bak");
		rename(filename, newfilename);
		}
	else
		exit(1);
	}

// bench.hpp
#define FALSE 0
#define TRUE !FALSE
#define NUMTIMES 10
#define MAXFILENAME 13

class Benchmark
	{
	public:
		Benchmark(char*);
		~Benchmark();
		void Click(void);
		void IterStart(int, char*);
		void IterStop(int, float);
		void LogCalcs(void);
	private:
		void Diff(struct timeb*, struct timeb*, struct timeb*);

		fstream logFile;
		fstream avgFile;
		char mapFileName[MAXFILENAME];
		struct timeb benchmarks[NUMTIMES][2];
		float distRobotTravelled[NUMTIMES];
		int timeTaken[NUMTIMES];
		int currentIter;
		int clickToggleFlag;
		struct timeb elapsedTime, computeTime;
		struct timeb startTime, clickOnTime, clickOffTime;
	};

// bench.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <alloc.h>		// for coreleft()
#include <stdlib.h>		// for itoa()
#include <string.h>		// for strcpy()
#include <sys\timeb.h>
#include "bench.hpp"


Benchmark::Benchmark(char *fileName)
	{
	char *tmp;
	char logname[MAXFILENAME];

	while((tmp = strchr(fileName, '\\')) != (char*)NULL)
		fileName = tmp+1;
	strcpy(logname, fileName);
```

```
	if ((tmp = strchr(logname, '.')) != (char*)NULL)
		strcpy(tmp, ".LOG");
	else
		strcat(logname, ".LOG");
	logFile.open(logname, ios::app);
	if (!logFile)
		cout << "Unable to open " << logname << "\n";

	if ((tmp = strchr(logname, '.')) != (char*)NULL)
		strcpy(tmp, ".AVG");
	else
		strcat(logname, ".AVG");
	avgFile.open(logname, ios::app);
	if (!avgFile)
		cout << "Unable to open " << logname << "\n";

	cout << "Initialised Benchmark class\n";
	}


Benchmark::~Benchmark()
	{
	if (logFile)
		{
		logFile.flush();
		logFile.close();
		}
	if (avgFile)
		{
		avgFile.flush();
		avgFile.close();
		}
	cout << "Closed log files and Destroying Benchmark class\n";
	}


void Benchmark::Click()
	{
	switch(clickToggleFlag)
		{
		case TRUE:
			ftime(&clickOnTime);
			clickToggleFlag = FALSE;
			break;
		case FALSE:
		default:
			ftime(&clickOffTime);
			Diff(&clickOnTime, &clickOffTime, &computeTime);
			clickToggleFlag = TRUE;
		}
	}


void Benchmark::Diff(struct timeb*start, struct timeb*stop, struct timeb*diff)
	{
	if ((*stop).millitm < (*start).millitm)
		{
		(*stop).millitm += (short)1000; /* carry when subtracting, stops*/
		(*start).time += 1L;            /* negative wraparound problems!*/
		}
	(*diff).millitm += (*stop).millitm - (*start).millitm;
	(*diff).time += (long)((*diff).millitm / (short)1000);
	(*diff).millitm %= (short)1000;
	(*diff).time += ((*stop).time-(*start).time);
	}


void Benchmark::IterStart(int i, char*s)
	{
	currentIter = i;
	clickToggleFlag = TRUE;
	computeTime.time = elapsedTime.time = 0L;
	computeTime.millitm = elapsedTime.millitm = 0;

	strcpy(mapFileName, s);

	ftime(&startTime);
	}


void Benchmark::IterStop(int t, float distTravelled)
	{
	struct timeb stopTime;
```

```
	ftime(&stopTime);
	Diff(&startTime, &stopTime, &elapsedTime);

	benchmarks[currentIter][0].time = elapsedTime.time;
	benchmarks[currentIter][0].millitm = elapsedTime.millitm;
	benchmarks[currentIter][1].time = computeTime.time;
	benchmarks[currentIter][1].millitm = computeTime.millitm;
	timeTaken[currentIter] = t;
	distRobotTravelled[currentIter] = distTravelled;
	}



void Benchmark::LogCalcs()
	{
	char tmp[256];
	float avgDist;
	struct timeb avg;
	int i, avgTimeTaken;

	for(i=0, avg.time=0L, avg.millitm=0, avgDist=0.0, avgTimeTaken=0;
		i<NUMTIMES;
		i++)
		{
		sprintf(tmp, "%s (%02d) Elapsed time:%05ld.%03d\n", mapFileName, i,
					benchmarks[i][0].time, benchmarks[i][0].millitm);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Compute time:%05ld.%03d\n", mapFileName, i,
					benchmarks[i][1].time, benchmarks[i][1].millitm);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Dist travelled:%f\n", mapFileName, i,
					distRobotTravelled[i]);
		logFile.write(tmp, strlen(tmp));
		sprintf(tmp, "%s (%02d) Time Slices Taken:%d\n", mapFileName, i,
					timeTaken[i]);
		logFile.write(tmp, strlen(tmp));
		avg.time+=benchmarks[i][0].time;
		avg.millitm+=benchmarks[i][0].millitm;
		if (avg.millitm % 1000 != avg.millitm)
			{
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;
			}
		}

	sprintf(tmp, "%s Tot. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));
	cout << avg.time << "." << avg.millitm << "\n";
	cout << tmp << "\n";

	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;

	sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	avgFile.write(tmp, strlen(tmp));
//	sprintf(tmp, "%s Avg. Elapsed time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));

	for(i=0, avg.time=0L, avg.millitm=0; i<NUMTIMES; i++)
		{
		avg.time+=benchmarks[i][1].time;
		avg.millitm+=benchmarks[i][1].millitm;
		if (avg.millitm % 1000 != avg.millitm)
			{
			avg.time += (long)(avg.millitm / (short)1000);
			avg.millitm %= (short)1000;
			}
		}
	sprintf(tmp, "%s Tot. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));
	i = (int) (avg.time % (long)NUMTIMES);
	avg.time /= (long)NUMTIMES;
	avg.millitm += i * 1000;
	avg.millitm /= NUMTIMES;
	sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	avgFile.write(tmp, strlen(tmp));
//	sprintf(tmp, "%s Avg. Compute time:%05ld.%03d\n", mapFileName, avg.time, avg.millitm);
	logFile.write(tmp, strlen(tmp));

	for(i=0, avgTimeTaken=0; i<NUMTIMES; i++)
		avgTimeTaken+=timeTaken[i];
```

```
        avgTimeTaken = avgTimeTaken / NUMTIMES;
        sprintf(tmp, "%s Avg. Time Taken:%d\n", mapFileName, avgTimeTaken);
        avgFile.write(tmp, strlen(tmp));
        logFile.write(tmp, strlen(tmp));

        for(i=0, avgDist=0.0; i<NUMTIMES; i++)
               avgDist += distRobotTravelled[i];
        avgDist = avgDist / ((float)NUMTIMES);
        sprintf(tmp, "%s Avg. Dist Travelled:%f\n", mapFileName, avgDist);
        avgFile.write(tmp, strlen(tmp));
        logFile.write(tmp, strlen(tmp));

        sprintf(tmp, "%s ===================\n", mapFileName);
        avgFile.write(tmp, strlen(tmp));
//      sprintf(tmp, "%s ===================\n", mapFileName);
        logFile.write(tmp, strlen(tmp));
        }

// domain.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00

#define FROM_FRONT 0
#define FROM_BACK 1
#define FROM_LEFT 2
#define FROM_RIGHT 3
#define FROM_FRONTRIGHT 4
#define FROM_FRONTLEFT 5
#define FROM_BACKRIGHT 6
#define FROM_BACKLEFT 7
#define FROM_UP 8
#define FROM_DOWN 9
#define FROM_NOWHERE 10

#define NOCOST (float)0.0
#define NORMAL (float)1.0
#define NORMAL_DIAG (float)1.414214
#define BLOCKED (float)1000.0

typedef struct
       {
       float dist;
       int from;
       char type;
       float cost[9];
       } POINT;

class Domain
       {
       public:
              Domain(int, int, int, char*);
              ~Domain();
              void AdvanceTime(void);
      float CheckLine(int, int, int, int, int, int);
              void DrawDomain(void);
      int GetDomainLength(void) { return(domainLength); }
      int GetDomainTimeSlices(void) { return(domainTimeSlices); }
      int GetDomainWidth(void) { return(domainWidth); }
              POINT* GetPoint(int, int, int);
      float GetPointCost(int, int, int, int);
              char GetPointFrom(int, int, int);
              char GetPointType(int, int, int);
              int IsPointClear(int, int, int);
              int IsPointGoal(int, int, int);
              int IsPointNearObject(int, int, int);
              int IsPointObject(int, int, int);
              int IsPointStart(int, int, int);
      int IsPointVertex(int, int, int);
              int MoveRobot(int, int, int, int, int, int, float*);
              void SetPointFrom(int, int, int, int);
```

```
        void SetPointType(int, int, int, char);
    private:
        void AgeTimeSlices(void);
    int CalcRobotDir(int, int, int, int, int, int);
        int ClearAdjPointOK(int, int, int);
        void ClearMobileObject(int, int, int);
        void ClearVerticesInTimeSlice(int);
        void DrawTimeSlice(int);
        void InitTimeSlice(int);
        void MarkMobileObject(int, int, int);
        void MoveMobileObject(OBJECT_NODE*);
        void MoveMobileObjects(void);
        void SetAdjObjsInTimeSlice(int);
        void SetGoalFromFile(char*);
        void SetMobileObjsFromFile(char*);
        void SetPermObjsInTimeSlice(int);
        void SetStartFromFile(char*);
        void SetVerticesInTimeSlice(int);

        ObjectList objList;
        POINT** domainHead;
    int domainWidth;
    int domainLength;
    int domainTimeSlices;
    };

// domain.cpp
#include <iostream.h>
#include <fstream.h>
#include <alloc.h>          // for coreleft()
#include <stdio.h>
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <conio.h>
#include "object.hpp"
#include "domain.hpp"

Domain::Domain(int numTimeSlices, int width, int length, char*mapFileName)
    {
    int i;

    cout << "Contructing Domain class\n";
    domainTimeSlices = numTimeSlices;
    domainWidth = width;
    domainLength = length;

    cout << "FreeHeap:" << farcoreleft() << "\n";
    cout << "Amount needed for one timeslice:";
    cout << width*length*sizeof(POINT) << "\n";

    domainHead = (POINT**)farcalloc(numTimeSlices, sizeof(POINT**));
    if (domainHead == (POINT**)NULL)
        {
        cout << "Not enough memory to build model of world\n";
        domainHead = (POINT***)NULL;
        domainTimeSlices = domainWidth = domainLength = 0;
        return;
        }

    for(i=0; i<numTimeSlices; i++)
        {
        domainHead[i] = (POINT*)farcalloc(width*length, sizeof(POINT));
        cout << "FreeHeap after timeslice allocated:" << farcoreleft() << "\n";
        if (domainHead[i] == (POINT*)NULL)
            {
            if (i==0)
                cout << "Not enough memory to build model of world\n";
            else
                cout << "Only enough memory to build " << i << " of the ";
                cout << numTimeSlices << " timeslices in model of world\n";
            domainTimeSlices = i;
            break;
            }
        InitTimeSlice(i);
        SetPermObjsInTimeSlice(i);
        }
    SetStartFromFile(mapFileName);
    SetGoalFromFile(mapFileName);
    SetMobileObjsFromFile(mapFileName);
    for(i=0; i<domainTimeSlices; i++)
        {
        SetAdjObjsInTimeSlice(i);
        SetVerticesInTimeSlice(i);
        }
```

```
        for(i=0; i<domainTimeSlices-1; i++)
                AdvanceTime();
        cout << "FreeHeap after Domain allocated:" << farcoreleft() << "\n";
        }



Domain::~Domain()
        {
        POINT *tmp;
        int i;

        cout << "Destructing Domain class\n";
        if (domainHead == (POINT**)NULL)
                {
                cout << "Not freeing domain - DomainHead was NULL\n";
                return;
                }
        for(i=0; i<domainTimeSlices; i++)
                {
                tmp = domainHead[i];
                farfree(tmp);
                }
        farfree(domainHead);
        domainHead = (POINT**)NULL;
        cout << "FreeHeap after Domain deallocated:" << farcoreleft() << "\n";
//      getch();
        }


void Domain::AdvanceTime()
        {
        AgeTimeSlices();
//      DrawDomain();
        ClearVerticesInTimeSlice(domainTimeSlices-1);
        MoveMobileObjects();
//      DrawDomain();
        SetVerticesInTimeSlice(domainTimeSlices-1);
//      DrawDomain();
        }


//////////////////////////////////////
//
// Copy the contents of every timeslice
// into the previous timeslice
// [0] <= [1], [1] <= [2], etc.
// Leave the last timeslice unchanged.
// Another routine will decide the moves
// for all the mobile objects.
//
// A quick way to do this is moving ptrs
// to the timeslices and only copying the
// contents of the last timeslice over the
// contents of the first timeslice
//
//////////////////////////////////////
void Domain::AgeTimeSlices()
        {
        int i, j;
        POINT *tmp;

        tmp = domainHead[0];
        for(i=0, j=1; i<domainTimeSlices-1; i++, j++)
                domainHead[i] = domainHead[j];
        domainHead[domainTimeSlices-1] = tmp;
        _fmemcpy(domainHead[domainTimeSlices-1],
                              domainHead[domainTimeSlices-2],
                              sizeof(*domainHead[domainTimeSlices-1]));
        }


int Domain::CalcRobotDir(int st, int sw, int sl, int et, int ew, int el)
        {
        if ((!IsPointStart(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(clear);

        if ((ew - sw > 0)
        && (el - sl > 0))
                {
                sw++, sl++;
                if ((IsPointClear(st, sw, sl))
                || (IsPointVertex(st, sw, sl))
```

```
        || (IsPointGoal(st, sw, sl)))
                return(frontright);
        else
                return(clear);
        }

if ((ew - sw > 0)
&& (el - sl == 0))
        {
        sw++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(right);
        else
                return(clear);
        }

if ((ew - sw > 0)
&& (el - sl < 0))
        {
        sw++, sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(backright);
        else
                return(clear);
        }

if ((ew - sw == 0)
&& (el - sl > 0))
        {
        sl++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(front);
        else
                return(clear);
        }

if ((ew - sw == 0)
&& (el - sl == 0)
&& (et - st > 0))
        {
        st++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(up);
        else
                return(clear);
        }

if ((ew - sw == 0)
&& (el - sl < 0))
        {
        sl--;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(back);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl > 0))
        {
        sw--, sl++;
        if ((IsPointClear(st, sw, sl))
        || (IsPointVertex(st, sw, sl))
        || (IsPointGoal(st, sw, sl)))
                return(frontleft);
        else
                return(clear);
        }

if ((ew - sw < 0)
&& (el - sl == 0))
        {
        sw--;
        if ((IsPointClear(st, sw, sl))
```

```
          || (IsPointVertex(st, sw, sl))
          || (IsPointGoal(st, sw, sl)))
                return(left);
          else
                return(clear);
          }

     if ((ew - sw < 0)
     && (el - sl < 0))
          {
          sw--, sl--;
          if ((IsPointClear(st, sw, sl))
          || (IsPointVertex(st, sw, sl))
          || (IsPointGoal(st, sw, sl)))
                return(backleft);
          else
                return(clear);
          }
     return(clear);
     }


float Domain::CheckLine(int st, int sw, int sl, int et, int ew, int el)
     {
//   static int displayCounter=1;
     int tmpt, tmpw, tmpl;
     float dist;

     for(dist=0.0, tmpt=et-st, tmpw=ew-sw, tmpl=el-sl;
          (tmpt != 0) || (tmpw != 0) || (tmpl != 0);
          tmpt=et-st, tmpw=ew-sw, tmpl=el-sl)
          {
          if (tmpt < 0)
                {
                return(-1.0);
                }
          if ((tmpw > 0)
          && (tmpl > 0))
                {
                dist += GetPointCost(st, sw++, sl++, frontright);
                if ((!IsPointClear(st, sw, sl))
          && ((st != et) || (sw != ew) || (sl != el)))
                        return(-1.0);
      continue;
                }
    if ((tmpw > 0)
          && (tmpl == 0))
                {
      dist += GetPointCost(st, sw++, sl, right);
          if ((!IsPointClear(st, sw, sl))
          && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
      continue;
                }
          if ((tmpw > 0)
          && (tmpl < 0))
                {
      dist += GetPointCost(st, sw++, sl--, backright);
                if ((!IsPointClear(st, sw, sl))
          && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
      continue;
      }
          if ((tmpw < 0)
          && (tmpl > 0))
                {
      dist += GetPointCost(st, sw--, sl++, frontleft);
          if ((!IsPointClear(st, sw, sl))
          && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
      continue;
      }
          if ((tmpw < 0)
          && (tmpl == 0))
                {
      dist += GetPointCost(st, sw--, sl, left);
          if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
      continue;
      }
          if ((tmpw < 0)
          && (tmpl < 0))
```

```
                {
        dist += GetPointCost(st, sw--, sl--, backleft);
                if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                        return(-1.0);
        continue;
                }
            if ((tmpw == 0)
            && (tmpl > 0))
                {
        dist += GetPointCost(st, sw, sl++, front);
            if ((!IsPointClear(st, sw, sl))
                && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
        continue;
        }
            if ((tmpw == 0)
            && (tmpl < 0))
                {
        dist += GetPointCost(st, sw, sl--, back);
            if ((!IsPointClear(st, sw, sl))
            && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
        continue;
        }
      if ((tmpw == 0)
            && (tmpl == 0)
            && (tmpt >0))
                {
        dist += GetPointCost(st++, sw, sl, up);
            if ((!IsPointClear(st, sw, sl))
            && ((st != et) || (sw != ew) || (sl != el)))
                return(-1.0);
        continue;
        }
            }
//      cout << "CheckLine:" << displayCounter++ << ":returned" << dist << "\n";
        return(dist);
        }


int Domain::ClearAdjPointOK(int t, int w, int l)
        {
        char type;
        int a,b;

        for(a=w-1;a<=w+1;a++)
                {
                if ((a < 0) || (a >= domainWidth))
                        continue;
                for(b=l-1;b<=l+1;b++)
                        {
                        if ((b < 0) || (b >= domainLength))
                                continue;
                        if ((a == w) && (b == l))
                                continue;

           type = GetPointType(t, a, b);
                        if ((type == OBJECT)
                        ||  (type == MOBILE_OBJECT))
                                return(FALSE);
                        }
                }
        return(TRUE);
        }


void Domain::ClearMobileObject(int t, int w, int l)
    {
        int i, j;

        if (GetPointType(t, w, l) == MOBILE_OBJECT)
                {
                if (ClearAdjPointOK(t, w, l))
                        {
                        SetPointType(t, w, l, CLEAR);
                        SetPointFrom(t, w, l, FROM_NOWHERE);
                        }
                else
                        {
                        SetPointType(t, w, l, ADJ_TO_OBJECT);
                        SetPointFrom(t, w, l, FROM_NOWHERE);
                        }
                }
```

```
	for(i=w-1, j=l-1; j<l+2;)
		{
	if ((GetPointType(t, i, j) == ADJ_TO_OBJECT)
		&& (ClearAdjPointOK(t, i, j)))
		{
		SetPointType(t, i, j, CLEAR);
			SetPointFrom(t, i, j, FROM_NOWHERE);
			}
		if (i == w+1)
			{
			i = w-1;
			j++;
			}
		else
			i++;
		}
	}


void Domain::ClearVerticesInTimeSlice(int t)
	{
	int w, l, pointType;

	for(w=0; w<domainWidth; w++)
		{
		for(l=0; l<domainLength; l++)
			{
			pointType = GetPointType(t, w, l);
			if (pointType == VERTEX)
				{
				SetPointFrom(t, w, l, FROM_NOWHERE);
				SetPointType(t, w, l, CLEAR);
				continue;
				}
			if (pointType == MOBILE_OBJECT)
				{
		ClearMobileObject(t, w, l);
				}
			}
		}
	}


void Domain::DrawDomain()
	{
	int i;

	for (i=0; i< domainTimeSlices; i++)
		{
		DrawTimeSlice(i);
//		getch();
		}
	}


void Domain::DrawTimeSlice(int timeSlice)
	{
	char type;
	int a,b,y;

	clrscr();
	gotoxy(1,1);
	cout << "time=t+" << timeSlice;

	for(b=0, y=2;b<domainLength; b++, y++)
		{
		gotoxy(1, y);
		for(a=0; a<domainWidth; a++)
			{
			type = GetPointType(timeSlice, a, b);
			switch(type)
				{
				case GOAL:
				case START:
				case ADJ_TO_OBJECT:
				case OBJECT:
				case MOBILE_OBJECT:
				case VERTEX:
					cout << type;
					break;
				case CLEAR:
					switch(GetPointFrom(timeSlice, a, b))
						{
						case FROM_RIGHT:
```

```
                                                        cout << "R";
                                                        break;
                                                case FROM_LEFT:
                                                        cout << "L";
                                                        break;
                                                case FROM_UP:
                                                        cout << "U";
                                                        break;
                                                case FROM_DOWN:
                                                        cout << "D";
                                                        break;
                                                case FROM_FRONT:
                                                        cout << "F";
                                                        break;
                                                case FROM_BACK:
                                                        cout << "B";
                                                        break;
                                                case FROM_BACKLEFT:
                                                        cout << "T";
                                                        break;
                                                case FROM_BACKRIGHT:
                                                        cout << "U";
                                                        break;
                                                case FROM_FRONTLEFT:
                                                        cout << "V";
                                                        break;
                                                case FROM_FRONTRIGHT:
                                                        cout << "W";
                                                        break;
                                                case FROM_NOWHERE:
                                                default:
                                                        cout << ".";
                                                        break;
                                                }
                                        break;
                                default:
                                        cout << "?";
                                        break;
                                }
                        }
                cout << "\n";
                }
        }


POINT* Domain::GetPoint(int timeSlice, int width, int length)
        {
    POINT *tmp, *tmp2;

    tmp = domainHead[timeSlice];
        tmp2 = tmp + (domainWidth*width) + length;
    return(tmp2);
    }


float Domain::GetPointCost(int timeSlice, int width, int length, int dir)
    {
    POINT *tmp;

    tmp = GetPoint(timeSlice, width, length);
        return(tmp->cost[dir]);
        }

char Domain::GetPointFrom(int timeSlice, int width, int length)
    {
    POINT *tmp;

    tmp = GetPoint(timeSlice, width, length);
        return(tmp->from);
    }


char Domain::GetPointType(int timeSlice, int width, int length)
        {
    POINT *tmp;

        tmp = GetPoint(timeSlice, width, length);
        return(tmp->type);
        }


void Domain::InitTimeSlice(int timeSlice)
        {
        POINT*tmp;
```

```
int a,b;

    for(a=0;a<domainWidth;a++)
        {
        for(b=0;b<domainLength;b++)
            {
    tmp = GetPoint(timeSlice, a, b);
            tmp->dist=0.0;
            tmp->from=FROM_NOWHERE;
            tmp->type=CLEAR;

/*          if (timeSlice == domainTimeSlices-1)
                tmp->cost[FROM_UP] = BLOCKED;
            else*/
                tmp->cost[FROM_UP] = NORMAL;

            if ((a == 0) && (b == 0))
                {
                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                tmp->cost[FROM_BACK] = BLOCKED;
                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                tmp->cost[FROM_LEFT] = BLOCKED;
                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
            if ((a == 0) && (b < domainLength-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                tmp->cost[FROM_LEFT] = BLOCKED;
                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
            if ((a == 0) && (b==domainLength-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                tmp->cost[FROM_LEFT] = BLOCKED;
                tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                tmp->cost[FROM_FRONT] = BLOCKED;
                continue;
                }
            if ((a < domainWidth-1) && (b == 0))
                {
                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                tmp->cost[FROM_BACK] = BLOCKED;
                tmp->cost[FROM_BACKLEFT] = BLOCKED;
                tmp->cost[FROM_LEFT] = NORMAL;
                tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
            if ((a < domainLength-1) && (b < domainWidth-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_LEFT] = NORMAL;
                tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_FRONT] = NORMAL;
                continue;
                }
            if ((a < domainWidth-1) && (b == domainLength-1))
                {
                tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                tmp->cost[FROM_RIGHT] = NORMAL;
                tmp->cost[FROM_BACKRIGHT] = NORMAL_DIAG;
                tmp->cost[FROM_BACK] = NORMAL;
                tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                tmp->cost[FROM_LEFT] = NORMAL;
```

```
                        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                        tmp->cost[FROM_FRONT] = BLOCKED;
                        continue;
                        }
                if ((a == domainWidth-1) && (b == 0))
                        {
                        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                        tmp->cost[FROM_RIGHT] = BLOCKED;
                        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                        tmp->cost[FROM_BACK] = BLOCKED;
                        tmp->cost[FROM_BACKLEFT] = BLOCKED;
                        tmp->cost[FROM_LEFT] = NORMAL;
                        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                        tmp->cost[FROM_FRONT] = NORMAL;
                        continue;
                        }
                if ((a == domainWidth-1) && (b < domainWidth-1))
                        {
                        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                        tmp->cost[FROM_RIGHT] = BLOCKED;
                        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                        tmp->cost[FROM_BACK] = NORMAL;
                        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                        tmp->cost[FROM_LEFT] = NORMAL;
                        tmp->cost[FROM_FRONTLEFT] = NORMAL_DIAG;
                        tmp->cost[FROM_FRONT] = NORMAL;
                        continue;
                        }
                if ((a == domainLength-1) && (b == domainWidth-1))
                        {
                        tmp->cost[FROM_FRONTRIGHT] = BLOCKED;
                        tmp->cost[FROM_RIGHT] = BLOCKED;
                        tmp->cost[FROM_BACKRIGHT] = BLOCKED;
                        tmp->cost[FROM_BACK] = NORMAL;
                        tmp->cost[FROM_BACKLEFT] = NORMAL_DIAG;
                        tmp->cost[FROM_LEFT] = NORMAL;
                        tmp->cost[FROM_FRONTLEFT] = BLOCKED;
                        tmp->cost[FROM_FRONT] = BLOCKED;
                        continue;
                        }
                }
        }
}


int Domain::IsPointClear(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case CLEAR:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }


int Domain::IsPointGoal(int t, int w, int l)
        {
        if ((w < 0)
        || (l < 0)
        || (t < 0)
        || (w >= domainWidth)
        || (l >= domainLength)
        || (t >= domainTimeSlices))
                return(FALSE);
        switch(GetPointType(t, w, l))
                {
                case GOAL:
                        return(TRUE);
                default:
                        return(FALSE);
                }
        }
```

```
int Domain::IsPointNearObject(int t, int w, int l)
	{
	if ((w < 0)
	|| (l < 0)
	|| (t < 0)
	|| (w >= domainWidth)
	|| (l >= domainLength)
	|| (t >= domainTimeSlices))
		return(TRUE);
	switch(GetPointType(t, w, l))
		{
		case OBJECT:
			return(OBJECT);
		case MOBILE_OBJECT:
			return(MOBILE_OBJECT);
		case ADJ_TO_OBJECT:
			return(ADJ_TO_OBJECT);
		default:
			return(FALSE);
		}
	}

int Domain::IsPointObject(int t, int w, int l)
	{
	if ((w < 0)
	|| (l < 0)
	|| (t < 0)
	|| (w >= domainWidth)
	|| (l >= domainLength)
	|| (t >= domainTimeSlices))
		return(TRUE);
	switch(GetPointType(t, w, l))
		{
		case OBJECT:
			return(OBJECT);
		case MOBILE_OBJECT:
			return(MOBILE_OBJECT);
		case ADJ_TO_OBJECT:
		default:
			return(FALSE);
		}
	}


int Domain::IsPointStart(int t, int w, int l)
	{
	if ((w < 0)
	|| (l < 0)
	|| (t < 0)
	|| (w >= domainWidth)
	|| (l >= domainLength)
	|| (t >= domainTimeSlices))
		return(FALSE);
	switch(GetPointType(t, w, l))
		{
		case START:
			return(TRUE);
		default:
			return(FALSE);
		}
	}


int Domain::IsPointVertex(int t, int w, int l)
	{
	if ((w < 0)
	|| (l < 0)
	|| (t < 0)
	|| (w >= domainWidth)
	|| (l >= domainLength)
	|| (t >= domainTimeSlices))
		return(FALSE);
	switch(GetPointType(t, w, l))
		{
		case VERTEX:
			return(TRUE);
		default:
			return(FALSE);
		}
	}
```

```
void Domain::MarkMobileObject(int t, int w, int l)
	{
	int i, j;

	if ((GetPointType(t, w, l) == CLEAR)
	|| (GetPointType(t, w, l) == ADJ_TO_OBJECT))
		{
		SetPointType(t, w, l, MOBILE_OBJECT);
		SetPointFrom(t, w, l, FROM_NOWHERE);
		}
	for(i=w-1, j=l-1; j<l+2;)
		{
		if (GetPointType(t, i, j) == CLEAR)
			{
			SetPointType(t, i, j, ADJ_TO_OBJECT);
			SetPointFrom(t, i, j, FROM_NOWHERE);
			}
		if (i == w+1)
			{
			i = w-1;
			j++;
			}
		else
			i++;
		}
	}


void Domain::MoveMobileObject(OBJECT_NODE*object)
	{
	int t=domainTimeSlices-1, w=(*object).w, l=(*object).l;

	ClearMobileObject(t, w, l);
	switch((*object).direction)
		{
		case front:
			if ((IsPointObject(t, w, l+1))
			|| (IsPointStart(t, w, l+1))
			|| (IsPointGoal(t, w, l+1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).l++;
			break;
		case frontleft:
			if ((IsPointObject(t, w-1, l+1))
			|| (IsPointStart(t, w-1, l+1))
			|| (IsPointGoal(t, w-1, l+1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).l++, (*object).w--;
			break;
		case left:
			if ((IsPointObject(t, w-1, l))
			|| (IsPointStart(t, w-1, l))
			|| (IsPointGoal(t, w-1, l)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w--;
			break;
		case backleft:
			if ((IsPointObject(t, w-1, l-1))
			|| (IsPointStart(t, w-1, l-1))
			|| (IsPointGoal(t, w-1, l-1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w--, (*object).l--;
			break;
		case back:
			if ((IsPointObject(t, w, l-1))
			|| (IsPointStart(t, w, l-1))
			|| (IsPointGoal(t, w, l-1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).l--;
			break;
		case backright:
			if ((IsPointObject(t, w+1, l-1))
			|| (IsPointStart(t, w+1, l-1))
			|| (IsPointGoal(t, w+1, l-1)))
				(*object).direction = rand() % NUM_DIRS;
			else
				(*object).w++, (*object).l--;
```

```
				break;
			case right:
				if ((IsPointObject(t, w+1, l))
				|| (IsPointStart(t, w+1, l))
				|| (IsPointGoal(t, w+1, l)))
					(*object).direction = rand() % NUM_DIRS;
				else
					(*object).w++;
				break;
			case frontright:
				if ((IsPointObject(t, w+1, l+1))
				|| (IsPointStart(t, w+1, l+1))
				|| (IsPointGoal(t, w+1, l+1)))
					(*object).direction = rand() % NUM_DIRS;
				else
					(*object).w++, (*object).l++;
				break;
			default:
				cout << "*** unknown direction for mobile object ignored ***\n";
				break;
			}
		MarkMobileObject(t, object->w, object->l);
		}


void Domain::MoveMobileObjects()
		{
		OBJECT_NODE*cur, *orig;

		orig = cur = objList.GetNextObject();
		if (cur == (OBJECT_NODE*)NULL)
		return;

		do
			{
			if ((*cur).velocity > 0)
				MoveMobileObject(cur);

		cur = objList.GetNextObject();
			} while (cur != orig);
		}


int Domain::MoveRobot(int st, int sw, int sl, int et, int ew, int el, float*dist)
		{
		int dir;

		*dist = 0.0;
		if (!IsPointStart(st, sw, sl))
			{
			*dist = 0.0;
			return(FALSE);
			}

		dir = CalcRobotDir(st, sw, sl, et, ew, el);
		SetPointType(st, sw, sl, CLEAR);
		SetPointFrom(st, sw, sl, FROM_NOWHERE);

		switch(dir)
			{
			case frontright:
				*dist = GetPointCost(st, sw, sl, frontright);
				st++, sw++, sl++;
				if ((!IsPointClear(st, sw, sl))
				&& (!IsPointVertex(st, sw, sl))
				&& (!IsPointGoal(st, sw, sl)))
					{
					st--, sw--, sl--;
					*dist = 0.0;
					}
				break;
			case right:
				*dist = GetPointCost(st, sw, sl, right);
				st++, sw++;
				if ((!IsPointClear(st, sw, sl))
				&& (!IsPointVertex(st, sw, sl))
				&& (!IsPointGoal(st, sw, sl)))
					{
					st--, sw--;
					*dist = 0.0;
					}
				break;
			case backright:
```

```
                *dist = GetPointCost(st, sw, sl, backright);
                st++, sw++, sl--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sw--, sl++;
                        *dist = 0.0;
                        }
                break;
        case frontleft:
                *dist = GetPointCost(st, sw, sl, frontleft);
                st++, sw--, sl++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sw++, sl--;
                        *dist = 0.0;
                        }
                break;
        case left:
                *dist = GetPointCost(st, sw, sl, left);
                st++, sw--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sw++;
                        *dist = 0.0;
                        }
                break;
        case backleft:
                *dist = GetPointCost(st, sw, sl, backleft);
                st++, sw--, sl--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sw++, sl++;
                        *dist = 0.0;
                        }
                break;
        case front:
                *dist = GetPointCost(st, sw, sl, front);
                st++, sl++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sl--;
                        *dist = 0.0;
                        }
                break;
        case back:
                *dist = GetPointCost(st, sw, sl, back);
                st++, sl--;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--, sl++;
                        *dist = 0.0;
                        }
                break;
        case up:
                *dist = GetPointCost(st, sw, sl, up);
                st++;
                if ((!IsPointClear(st, sw, sl))
                && (!IsPointVertex(st, sw, sl))
                && (!IsPointGoal(st, sw, sl)))
                        {
                        st--;
                        *dist = 0.0;
                        }
                break;
        default:        // no movement - start IS goal
                *dist = 0.0;
                break;
        }
if (IsPointGoal(st, sw, sl))
        return(TRUE);
else
        {
```

```
			SetPointType(st, sw, sl, START);
			SetPointFrom(st, sw, sl, FROM_NOWHERE);
			return(FALSE);
			}
	}


void Domain::SetAdjObjsInTimeSlice(int timeSlice)
	{
	int a,b;
	int w,l;

	for(w=0; w<domainWidth; w++)
		{
		for(l=0; l<domainLength; l++)
			{
			if (!IsPointObject(timeSlice, w, l))
				continue;

			for(a=w-1; a<=w+1; a++)
				{
				if ((a < 0) || (a >= domainWidth))
					continue;
				for(b=l-1; b<=l+1; b++)
					{
					if ((b < 0) || (b >= domainLength))
						continue;

					if (GetPointType(timeSlice, a, b) == CLEAR)
						{
						SetPointFrom(timeSlice, a, b, FROM_NOWHERE);
						SetPointType(timeSlice, a, b, ADJ_TO_OBJECT);
						}
					}
				}
			}
		}
	}


void Domain::SetGoalFromFile(char*fileName)
	{
	int i, w, l;
	char recType;
	char tmp[256];
	fstream dataFile;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Goal added.\n";

	for(;;)
		{
		dataFile. getline(tmp, sizeof(tmp));
		if (strlen(tmp) == 0)
			break;
		if (tmp[0] == GOAL)
			{
//			cout << "The GOAL line is: " << tmp << "\n";
			if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
				{
	    for(i=0; i<domainTimeSlices; i++)
	      {
				SetPointFrom(i, w, l, FROM_NOWHERE);
				SetPointType(i, w, l, GOAL);
	      }
				break;
				}
			else
				cout << "Improperly formatted line ignored\n";
			}
		}
	dataFile.close();
	}


void Domain::SetMobileObjsFromFile(char*fileName)
	{
	int w, l;
	char recType;
	char tmp[256];
	fstream dataFile;
	OBJECT_NODE* newObj;
```

```
        dataFile.open(fileName, ios::in);
        if (!dataFile)
                cout << "Unable to open " << fileName << ". No Mobile objs added.\n";

        for(;;)
                {
                int i;

                dataFile.getline(tmp, sizeof(tmp));
                i = strlen(tmp);
                if (i <= 0)
                        break;
                cout << "Length of input line is: " << i << "\n";
                if ((tmp[0] != GOAL)
                && (tmp[0] != START))
                        {
//                      cout << "OBJECT line is: " << tmp << "\n";
                        if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
                                {
                                SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
                                SetPointType(domainTimeSlices-1, w, l, MOBILE_OBJECT);
                                newObj = objList.BuildNewObject(w, l);
                                if (newObj)
                                        {
                                        objList.InsertNewObject(newObj);
//                                      cout << "Added obj to obj list\n";
                                        }
                                }
                        else
                                cout << "Improperly formatted line ignored\n";
                        }
                }
        dataFile.close();
        }


void Domain::SetPermObjsInTimeSlice(int timeSlice)
        {
        int w, l;

        for(w=0; w<domainWidth-5; w++)
                {
                l=2;
                switch(w)
                        {
                        case 3:
                        case 4:
                        case 5:
                                break;
                        default:
                                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                                SetPointType(timeSlice, w, l, OBJECT);
//                              SetAdjObjsInTimeSlice(timeSlice, w, l);
                                break;
                        }
                }
        for(w=0; w<domainWidth-10; w++)
                {
                l=7;
                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                SetPointType(timeSlice, w, l, OBJECT);
//              SetAdjObjsInTimeSlice(timeSlice, w, l);
                }
        for(l=0, w=domainWidth-10; l<domainLength-2; l++)
                {
                switch(l)
                        {
                        case 3:
                        case 4:
                        case 5:
                        case 6:
                        case 14:
                        case 15:
                        case 16:
                                break;
                        default:
                                SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
                                SetPointType(timeSlice, w, l, OBJECT);
//                              SetAdjObjsInTimeSlice(timeSlice, w, l);
                                break;
                        }
                }
        for(l=2, w=domainWidth-5; l<domainLength-7; l++)
                {
```

```
		switch(l)
			{
			case 9:
			case 10:
			case 11:
				break;
			default:
				SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
				SetPointType(timeSlice, w, l, OBJECT);
//				SetAdjObjsInTimeSlice(timeSlice, w, l);
				break;
			}
		}
	for(w=domainWidth-5, l=domainLength-7; w<domainWidth; w++)
		{
		SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
		SetPointType(timeSlice, w, l, OBJECT);
//		SetAdjObjsInTimeSlice(timeSlice, w, l);
		}
	for(w=domainWidth-10, l=domainLength-2; w<domainWidth; w++)
		{
		SetPointFrom(timeSlice, w, l, FROM_NOWHERE);
		SetPointType(timeSlice, w, l, OBJECT);
//		SetAdjObjsInTimeSlice(timeSlice, w, l);
		}
	}


void Domain::SetPointFrom(int timeSlice, int width, int length, int from)
	{
	POINT *tmp;

	if ((width < 0)
	|| (length < 0)
	|| (timeSlice < 0)
	|| (width >= domainWidth)
	|| (length >= domainLength)
	|| (timeSlice >= domainTimeSlices))
		return;
	tmp = GetPoint(timeSlice, width, length);
	tmp->from = from;
	}

void Domain::SetPointType(int timeSlice, int width, int length, char type)
	{
	char oldType;
	POINT *tmp;

	if ((width < 0)
	|| (length < 0)
	|| (timeSlice < 0)
	|| (width >= domainWidth)
	|| (length >= domainLength)
	|| (timeSlice >= domainTimeSlices))
		return;
	tmp = GetPoint(timeSlice, width, length);
	oldType = tmp->type;
	tmp->type = type;
	if (oldType != OBJECT)
		{
//		SetAdjObjsInTimeSlice(timeSlice, width, length);
		}
	}


void Domain::SetStartFromFile(char*fileName)
	{
	int w, l;
	char recType;
	char tmp[256];
	fstream dataFile;

	dataFile.open(fileName, ios::in);
	if (!dataFile)
		cout << "Unable to open " << fileName << ". No Start added.\n";

	for(;;)
		{
		dataFile. getline(tmp, sizeof(tmp));
		if (strlen(tmp) == 0)
			break;
		if (tmp[0] == START)
			{
//			cout << "The START line is: " << tmp << "\n";
```

```
				if (sscanf(tmp, "%c%d%d", &recType, &w, &l) == 3)
					{
					SetPointFrom(domainTimeSlices-1, w, l, FROM_NOWHERE);
					SetPointType(domainTimeSlices-1, w, l, START);
					break;
					}
				else
					cout << "Improperly formatted line ignored\n";
				}
			}
		dataFile.close();
		}


void Domain::SetVerticesInTimeSlice(int t)
		{
		int w, l;
		int diff_counter, corner_counter;
		int side_bits, corner_bits;

		for(w=0; w<domainWidth; w++)
			{
			for(l=0; l<domainLength; l++)
				{
				if (!IsPointClear(t, w, l))
					continue;

				corner_bits = BITMASK_CLEAR;
				corner_counter = 0;
				if (IsPointNearObject(t, w-1, l-1))
					{
					corner_bits |= BITMASK_LEFT;
					corner_bits |= BITMASK_TOP;
					corner_counter++;
					}
				if (IsPointNearObject(t, w-1, l+1))
					{
					corner_bits |= BITMASK_LEFT;
					corner_bits |= BITMASK_BOTTOM;
					corner_counter++;
					}
				if (IsPointNearObject(t, w+1, l+1))
					{
					corner_bits |= BITMASK_RIGHT;
					corner_bits |= BITMASK_BOTTOM;
					corner_counter++;
					}
				if (IsPointNearObject(t, w+1, l-1))
					{
					corner_bits |= BITMASK_RIGHT;
					corner_bits |= BITMASK_TOP;
					corner_counter++;
					}
				if (corner_bits != BITMASK_CLEAR)
					{
					side_bits = BITMASK_CLEAR;
					if (IsPointNearObject(t, w-1, l))
						side_bits |= BITMASK_LEFT;
					if (IsPointNearObject(t, w, l-1))
						side_bits |= BITMASK_TOP;
					if (IsPointNearObject(t, w+1, l))
						side_bits |= BITMASK_RIGHT;
					if (IsPointNearObject(t, w, l+1))
						side_bits |= BITMASK_BOTTOM;

					for(diff_counter=0;
						(side_bits != BITMASK_CLEAR)
						|| (corner_bits != BITMASK_CLEAR);
						side_bits >>= 1, corner_bits >>= 1)
						{
						if ((corner_bits & (int)0x01)
						&& ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
							{
							diff_counter++;
							continue;
							}
						if ((side_bits & (int)0x01)
						&& ((side_bits & (int)0x01) != (corner_bits & (int)0x01)))
							{
							diff_counter++;
							continue;
							}
						}
					if ((diff_counter > 2)
```

```
                                || ((diff_counter == 2) && (corner_counter == 1)))
                                        {
                                        SetPointFrom(t, w, l, FROM_NOWHERE);
                                        SetPointType(t, w, l, VERTEX);
                                        }
                                }
                        }
                }
        }

// edge.hpp
#define DEBUG_FILENAME "DEBUG.LOG"

typedef struct van
    {
        int t,w,l;
        float dist;
        struct van *prev, *next;
        } EDGE_NODE;

class EdgeList
        {
        public:
                EdgeList();
                ~EdgeList();
        EDGE_NODE* BuildNewEdge(int, int, int, float);
                void DelAllEdges(void);
                void DelEdge(EDGE_NODE*);
                void DelEdgeToVertex(int, int, int);
        EDGE_NODE* GetFirstEdge(void);
        EDGE_NODE* GetNextEdge(EDGE_NODE*);
        void InsertNewEdge(EDGE_NODE*);
                void ListAllEdges(int);
    private:
        EDGE_NODE *edgeHead;
        };

// edge.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include "edge.hpp"


EdgeList::EdgeList()
        {
        edgeHead = (EDGE_NODE*)NULL;
//    cout << "EdgeList Constructor\n";
        }


EdgeList::~EdgeList()
        {
        while (edgeHead != (EDGE_NODE*)NULL)
                DelEdge(edgeHead);
//      cout << "EdgeList Destructor\n";
        }


EDGE_NODE* EdgeList::BuildNewEdge(int t, int w, int l, float dist)
        {
        EDGE_NODE *a;

//      if (otherVertex != (void*)NULL)
//              {
                a = (EDGE_NODE*)farcalloc(1, sizeof(EDGE_NODE));
                if (a == (EDGE_NODE*)NULL)
                        {
                        cout << "Out of Memory in BuildNewEdge()\n";
          getch();
                        exit(0);
                        }
                (*a).t = t;
                (*a).w = w;
                (*a).l = l;
                (*a).dist = dist;
                (*a).prev = (*a).next = (EDGE_NODE*)NULL;
                return(a);
//              }
//      return((EDGE_NODE*)NULL);
```

```
        }

void EdgeList::DelAllEdges()
        {
        EDGE_NODE*a;

        for(a = GetFirstEdge();
                a != (EDGE_NODE*)NULL;
                a = GetFirstEdge())
                {
                DelEdge(a);
                }
        }

void EdgeList::DelEdge(EDGE_NODE*a)
        {
        EDGE_NODE*tmp;

//      cout << "edge 0\n";
        if (a != (EDGE_NODE*)NULL)
                {
                if ((a->prev == (EDGE_NODE*)NULL)  // del last remaining edge
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 1\n";
                        free(a);
                        edgeHead = (EDGE_NODE*)NULL;
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge in middle
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 2\n";
                        tmp = a->prev;
                        tmp->next = a->next;
                        tmp = a->next;
                        tmp->prev = a->prev;
                        free(a);
                        return;
                        }
                if ((a->prev == (EDGE_NODE*)NULL)  // del edge at sol
                && (a->next != (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 3\n";
                        tmp = a->next;
                        tmp->prev = (EDGE_NODE*)NULL;
                        edgeHead = tmp;
                        free(a);
                        return;
                        }
                if ((a->prev != (EDGE_NODE*)NULL)  // del edge at eol
                && (a->next == (EDGE_NODE*)NULL))
                        {
//                      cout << "edge 4\n";
                        tmp = a->prev;
                        tmp->next = (EDGE_NODE*)NULL;
                        free(a);
                        return;
                        }
                }
        }


void EdgeList::DelEdgeToVertex(int t, int w, int l)
        {
        EDGE_NODE*tmp;

        for(tmp = GetFirstEdge(); tmp != (EDGE_NODE*)NULL; tmp = GetNextEdge(tmp))
                {
                if ((tmp->t == t)
        && (tmp->w == w)
        && (tmp->l == l))
                        {
                        DelEdge(tmp);
                        break;
                        }
                }
        }


EDGE_NODE* EdgeList::GetFirstEdge()
        {
    return(edgeHead);
```

```
	}

EDGE_NODE* EdgeList::GetNextEdge(EDGE_NODE*cur)
	{
	return(cur->next);
	}


void EdgeList::InsertNewEdge(EDGE_NODE *edge)
		{
		EDGE_NODE*cur;

		if (edge == (EDGE_NODE*)NULL)
			return;

		if (edgeHead == (EDGE_NODE*)NULL)
			{
			edgeHead = edge;
			edge->prev = edge->next = (EDGE_NODE*)NULL;
//			cout << "Inserted edge into empty list ";
//			cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//			cout << "Dist = " << edge->dist << "\n";
			return;
			}

		for(cur=edgeHead; cur != (EDGE_NODE*)NULL; cur=(*cur).next)
			{
			if ( (edge->t > cur->t)
			|| ((edge->t == cur->t)
				&& (edge->w > cur->w))
			|| ((edge->t == cur->t)
				&& (edge->w == cur->w)
				&& (edge->l > cur->l)) )
				{					// insert after cur
				if (cur->next == (EDGE_NODE*)NULL)
					{
					edge->next = (EDGE_NODE*)NULL;// no more so append to eol
					edge->prev = cur;
					cur->next = edge;
					break;
					}
				else
					continue;				// try next one
				}
			if ( (edge->t < cur->t)
			|| ((edge->t == cur->t)
				&& (edge->w < cur->w))
/*			|| ((edge->t < cur->t)*/
			|| ((edge->t == cur->t)
				&& (edge->w == cur->w)
				&& (edge->l < cur->l)) )
				{					// insert before cur
				if (cur->prev == (EDGE_NODE*)NULL)
					{
					edge->prev = (EDGE_NODE*)NULL;	// at start of list
					edge->next = cur;
					cur->prev = edge;
					edgeHead = edge;
					break;
					}
				else
					{
					edge->prev = cur->prev;		// in middle/end of list
					edge->next = cur;
					cur->prev = edge;
					cur = edge->prev;
					cur->next = edge;
					break;
					}
				}
			if ( (edge->t == cur->t)
			&& (edge->w == cur->w)
			&& (edge->l == cur->l) )
				{
				/* already here - replace it ! */
/*				cout << "Already here - replacing values and disposing of new edge ";
				cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
				cout << "Dist = " << edge->dist << "\n";
				cur->w = edge->w;
				cur->l = edge->l;
				cur->dist = edge->dist;*/
//				cout << " Already here - ignoring...\n";
				cout << "!";
```

```
                        free(edge);
                        break;
                        }
                }
//      cout << "Inserted edge into list ";
//      cout << " (" << edge->t << "," << edge->w << "," << edge->l << "). ";
//      cout << "Dist = " << edge->dist << "\n";
        }


void EdgeList::ListAllEdges(int debugFlag)
        {
        char tmp[256];
        fstream debugFile;
        EDGE_NODE *edge;

        strcpy(tmp, "List of all edges in list\n");
        cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        for(edge = edgeHead; edge != (EDGE_NODE*)NULL; edge = (*edge).next)
                {
                sprintf(tmp, "Edge to (%d,%d,%d). Dist is %f\n",
                                                (*edge).t, (*edge).w, (*edge).l, (*edge).dist);
                cout << tmp;
                if (debugFlag)
                        {
                        debugFile.open(DEBUG_FILENAME, ios::app);
                        debugFile.write(tmp, strlen(tmp));
                        debugFile.close();
                        }
                }
        strcpy(tmp, "-----------------\n");
        cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        }

// object.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'

#define NUM_DIRS 7 /* this is the 8 horizontal directions; (0->7) */
enum directions { front, back, left, right, frontright, frontleft, backright, backleft, up, down, clear};


typedef struct o
        {
        int direction, velocity;
        int w, l;
        struct o *prev, *next;
        } OBJECT_NODE;


class ObjectList
        {
        public:
                ObjectList();
                ~ObjectList();
                OBJECT_NODE* BuildNewObject(int, int);
        void DelAllObjects(void);
        void DelObject(OBJECT_NODE*);
        OBJECT_NODE* GetNextObject(void);
                void InsertNewObject(OBJECT_NODE*);
        void ListAllObjects(void);
        private:
                OBJECT_NODE *objectHead;
```

```
	};

// object.cpp
#include <iostream.h>
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
#include <string.h>         // for strcpy()
#include <conio.h>
// #include "edge.hpp"
#include "object.hpp"

ObjectList::ObjectList()
	{
	objectHead = (OBJECT_NODE*)NULL;
	cout << "ObjectList Constructor\n";
	}


ObjectList::~ObjectList()
	{
	cout << "ObjectList Destructor\n";
	DelAllObjects();
	}


OBJECT_NODE* ObjectList::BuildNewObject(int w, int l)
	{
	OBJECT_NODE *newPtr;

	newPtr = (OBJECT_NODE*)farcalloc(1, sizeof(OBJECT_NODE));
	if (newPtr == (OBJECT_NODE*)NULL)
		{
		cout << "Out of memory in BuildNewObject()\n";
      getch();
		return(NULL);
		}
	(*newPtr).direction = rand() % NUM_DIRS;
	(*newPtr).velocity = 1;
	(*newPtr).prev = (*newPtr).next = (OBJECT_NODE*)NULL;
	(*newPtr).w = w;
	(*newPtr).l = l;
	return(newPtr);
	}


void ObjectList::DelAllObjects()
	{
	OBJECT_NODE *tmp;

	while (objectHead != (OBJECT_NODE*)NULL)
		{
		tmp = objectHead;
		objectHead = (*objectHead).next;
		DelObject(tmp);
		}
	}


void ObjectList::DelObject(OBJECT_NODE *todie)
   {
	OBJECT_NODE *cur;

	if (((*todie).prev != (OBJECT_NODE*)NULL)
	&& ((*todie).next != (OBJECT_NODE*)NULL))
		{
//		cout << "Deleted Object (" << todie->w << "," << todie->l << ")\n";
		cur=(*todie).prev;
		if (cur == todie)
			{
			objectHead = (OBJECT_NODE*)NULL;
//			cout << "Object List Empty\n";
			free(todie);
			return;
			}
		else
			{
			(*cur).next=(*todie).next;
			cur = (*todie).next;
			(*cur).prev = (*todie).prev;
			if (objectHead == todie)
```

```
                              objectHead = (*todie).next;
        free(todie);
          return;
                        }
              }
        cout << "***Did NOT delete rotton Object (" << todie->w << "," << todie->l << ")\n";
        }


OBJECT_NODE* ObjectList::GetNextObject()
    {
    OBJECT_NODE*tmp;

        if (objectHead == (OBJECT_NODE*)NULL)
               return((OBJECT_NODE*)NULL);

        tmp = objectHead;
    objectHead = objectHead->next;
    return(tmp);
    }


void ObjectList::InsertNewObject(OBJECT_NODE*newPtr)
        {
        OBJECT_NODE *cur;

        if (newPtr == (OBJECT_NODE*)NULL)
               return;

        if (objectHead == (OBJECT_NODE*)NULL)
               {
               objectHead = newPtr;
               (*newPtr).prev = (*newPtr).next = newPtr;
               }
        else
               {
               /* insert before first node */
               cur = objectHead;
               (*newPtr).next = cur;
               (*newPtr).prev = (*cur).prev;
               (*cur).prev = newPtr;

               cur = (*newPtr).prev;
               (*cur).next = newPtr;
               objectHead = newPtr;
        }
//      cout << "Inserted object (" << newPtr->w << "," << newPtr->l << ")\n";
        }



void ObjectList::ListAllObjects()
        {
   OBJECT_NODE*tmp;

   if (objectHead == (OBJECT_NODE*)NULL)
      {
      cout << "ObjectList is empty\n";
      return;
      }
   for(tmp=objectHead;;tmp = tmp->next)
      {
       cout << "Object:(" << tmp->w << "," << tmp->l << ")\n";
      tmp = tmp->next;
      if (tmp == objectHead)
         {
         cout << "ObjectList ended\n";
         break;
         }
      }
       }

// vertex.hpp
#define FALSE 0
#define TRUE !FALSE

#define GOAL 'G'
#define START 'S'
#define OBJECT 'X'
#define ADJ_TO_OBJECT 'x'
#define MOBILE_OBJECT 'M'
#define CLEAR '.'
#define VERTEX 'V'
```

```
#define BITMASK_TOP (int)0x08
#define BITMASK_LEFT (int)0x04
#define BITMASK_BOTTOM (int)0x02
#define BITMASK_RIGHT (int)0x01
#define BITMASK_CLEAR (int)0x00

#define DEBUG_FILENAME "DEBUG.LOG"

typedef struct vn
	{
	int t,w,l;
	char nodeType;
	struct vn *searchPrev, *searchNext;
	float searchDist, estDist;
	int searchMarker;
	EdgeList *edgeList;
	struct vn *pathFrom, *pathTo;
	struct vn *prev, *next;
	} VERTEX_NODE;


class VertexList
	{
	public:
		VertexList();
		~VertexList();
		void AddToSearchList(VERTEX_NODE*);
	VERTEX_NODE* BuildNewVertex(int,int,int,char);
		float CalcEstimatedDist(int,int,int,int);
		int CalcRobotDir(VERTEX_NODE*);
		void DelAllVertices(void);
	void DelVertex(VERTEX_NODE*);
	int FindPath(void);
	VERTEX_NODE* FindVertex(int, int, int);
		VERTEX_NODE* GetFirstVertex(void);
		VERTEX_NODE* GetGoalVertex(void);
		VERTEX_NODE* GetNextVertex(VERTEX_NODE*);
		int GetSearchMarker(void);
	VERTEX_NODE* GetStartVertex(void);
	void InsertAllVertices(void);
		void InsertNewVertex(VERTEX_NODE*);
		void ListAllVertices(int);
	void ListSearchList(int);
//		void MoveRobot(int, int, int, int);
		void MarkPath(VERTEX_NODE*);
		void RemoveFromSearchList(VERTEX_NODE*);
	void RemoveVertex(VERTEX_NODE*);
		void TrimSearchList(void);
	private:
		VERTEX_NODE* vertexHead;
	VERTEX_NODE* searchHead;
	int searchMarker;
	float searchTrimDist;
	};

// vertex.cpp
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <conio.h>		// for getch()
#include <alloc.h>		// for coreleft()
#include <stdlib.h>		// for itoa()
#include <string.h>		// for strcpy()
#include "edge.hpp"
#include "vertex.hpp"

VertexList::VertexList()
	{
	vertexHead = (VERTEX_NODE*)NULL;
   searchHead = (VERTEX_NODE*)NULL;
   searchMarker = 0;
   searchTrimDist = -1.0;
   cout << "Initialised Vertex class\n";
	}


VertexList::~VertexList()
	{
	cout << "VertexList destructor started\n";
	DelAllVertices();
	cout << "VertexList destructor ended!\n";
	}
```

```
void VertexList::AddToSearchList(VERTEX_NODE*a)
        {
        VERTEX_NODE *tmp;

        if (a == (VERTEX_NODE*)NULL)
                return;

        if (((*a).searchPrev != (VERTEX_NODE*)NULL) // if already in fringe
        || ((*a).searchNext != (VERTEX_NODE*)NULL)) // list, dont add again
                return;

        if (searchHead == (VERTEX_NODE*)NULL)
                {
                searchHead = a;
                a->pathFrom = a->pathTo = (VERTEX_NODE*)NULL;
                a->searchPrev = a->searchNext = a;
       return;
                }

    for (tmp = searchHead;tmp != (VERTEX_NODE*)NULL;)
       {
       if (a->estDist <= tmp->estDist)
          {
          a->searchPrev = tmp->searchPrev;
          a->searchNext = tmp;
          tmp->searchPrev = a;
          tmp = a->searchPrev;
          tmp->searchNext = a;
          if (searchHead == a->searchNext)
             searchHead = a;
          return;
          }
       if (tmp->searchNext == searchHead)
          {
          a->searchPrev = tmp;
          a->searchNext = searchHead;
                    tmp->searchNext = searchHead->searchPrev = a;
          return;
          }
       else
          tmp = tmp->searchNext;
       }
        }


VERTEX_NODE* VertexList::BuildNewVertex(int t, int w, int l, char nodeType)
        {
        VERTEX_NODE *newPtr;

        newPtr = (VERTEX_NODE*)farcalloc(1, sizeof(VERTEX_NODE));
        if (newPtr == (VERTEX_NODE*)NULL)
                {
                cout << "Farcalloc() failed in BuildNewVertex!\n";
       getch();
                exit(0);
                }
        (*newPtr).t = t;
        (*newPtr).w = w;
        (*newPtr).l = l;
        (*newPtr).nodeType = nodeType;
        (*newPtr).edgeList = new EdgeList();
        (*newPtr).pathTo = (*newPtr).pathFrom = (VERTEX_NODE*)NULL;
        (*newPtr).searchDist = 0.0;
   (*newPtr).estDist = 0.0;
        (*newPtr).searchPrev = (*newPtr).searchNext = (VERTEX_NODE*)NULL;
        (*newPtr).prev = (*newPtr).next = (VERTEX_NODE*)NULL;
        if (nodeType == START)
                {
                (*newPtr).searchMarker = searchMarker+1;
                AddToSearchList(newPtr);
                }
        else
                (*newPtr).searchMarker = searchMarker;
        return(newPtr);
        }


float VertexList::CalcEstimatedDist(int sw, int sl, int ew, int el)
   {
   int a;

   a = abs(sw - ew) + abs(sl - el);
   a/=2;
```

```
   return((float)a);
   }


void VertexList::DelAllVertices()
   {
       while (vertexHead != (VERTEX_NODE*)NULL)
             {
//           cout << "About to delete (" << vertexHead->t << "," << vertexHead->w ;
//           cout << "," << vertexHead->l << ") at addr: " << vertexHead ;
//           cout << ": Prev:" << vertexHead->prev << ": Next:" << vertexHead->next <<"\n";
             DelVertex(vertexHead);
             }
       cout << "----------\n";
//     getch();
       }


void VertexList::DelVertex(VERTEX_NODE *todie)
       {
       VERTEX_NODE *tmp;
       EDGE_NODE*a;

       if (todie == (VERTEX_NODE*)NULL)
              return;

       for(a = todie->edgeList->GetFirstEdge();
              a != (EDGE_NODE*)NULL;
              a = todie->edgeList->GetFirstEdge())
              {
      todie->edgeList->DelEdgeToVertex(a->t, a->w, a->l);
      tmp = FindVertex(a->t, a->w, a->l);
      if (tmp != (VERTEX_NODE*)NULL)
         {
         tmp->edgeList->DelEdgeToVertex(todie->t, todie->w, todie->l);
         }
//            tmp = (VERTEX_NODE*)(a->otherVertex);
//            cout << "On vertex (" << todie->t << "," << todie->w << ",";
//            cout << todie->l << "):Del edge to (" << tmp->t << "," << tmp->w << ",";
//            cout << tmp->l << ")\n";
//            cout << "todie at:" << todie << ". a at:" << a << "\n";
//            cout << "  tmp at:" << tmp << "\n";
//            todie->edgeList->DelEdgeToVertex((void*)tmp);
//            cout << "On vertex (" << tmp->t << "," << tmp->w << ",";
//            cout << tmp->l << "):Del edge to (" << todie->t << "," << todie->w << ",";
//            cout << todie->l << ")\n";
//            tmp->edgeList->DelEdgeToVertex((void*)todie);
              }
      delete todie->edgeList;
      RemoveFromSearchList(todie);

      if ( ((*todie).prev == (VERTEX_NODE*)NULL)
      && ((*todie).next == (VERTEX_NODE*)NULL) )
             {
             vertexHead = (VERTEX_NODE*)NULL;
             free(todie);
             return;
             }
      if ( ((*todie).prev != (VERTEX_NODE*)NULL)
      && ((*todie).next != (VERTEX_NODE*)NULL) )
             {
             tmp = (*todie).prev;
             (*tmp).next = (*todie).next;
             tmp = (*todie).next;
             (*tmp).prev = (*todie).prev;
             free(todie);
             return;
             }
      if ( ((*todie).prev == (VERTEX_NODE*)NULL)
      && ((*todie).next != (VERTEX_NODE*)NULL) )
             {
             vertexHead = tmp = (*todie).next;
             (*tmp).prev = (VERTEX_NODE*)NULL;
             free(todie);
             return;
             }
      if ( ((*todie).prev != (VERTEX_NODE*)NULL)
      && ((*todie).next == (VERTEX_NODE*)NULL) )
             {
             tmp = (*todie).prev;
             (*tmp).next = (VERTEX_NODE*)NULL;
             free(todie);
             return;
```

```
            }
        }


 int VertexList::FindPath()
    {
        EDGE_NODE *e;
        int goalFound=FALSE;
        float dist;
        VERTEX_NODE *cur, *adj, *dest, *t1, *t2;

        if (searchHead->nodeType == START)
              searchMarker = searchHead->searchMarker;
        else
              return(FALSE);

        t1 = GetStartVertex();
        t2 = GetGoalVertex();
        if ((t1 != (VERTEX_NODE*)NULL)
        && (t2 != (VERTEX_NODE*)NULL)
        && (t1->w == t2->w)
        && (t1->l == t2->l))
              {
              t1->pathTo = t2;
              return(TRUE);
              }

        for (cur = searchHead; cur != (VERTEX_NODE*)NULL; cur = searchHead)
              {
              if ((cur->searchDist >= searchTrimDist)
              && (searchTrimDist > 0.0))
                    {
                    RemoveFromSearchList(cur);
                    continue;
                    }
              for(e = cur->edgeList->GetFirstEdge();
                    e != (EDGE_NODE*)NULL;
                    e = cur->edgeList->GetNextEdge(e))
                    {
          adj = FindVertex(e->t, e->w, e->l);
          if (adj == (VERTEX_NODE*)NULL)
             continue;

//        adj = (VERTEX_NODE*) e->otherVertex;
                    dist = cur->searchDist + e->dist;
                    if ( (adj->searchMarker != searchMarker)
                    || ((adj->searchMarker == searchMarker)
                          && (adj->searchDist > dist)) )
                          {
                          if ((dist <= searchTrimDist)
                          || (searchTrimDist <= 0.0))
                                {
                                adj->pathFrom = cur;
                                adj->searchMarker = searchMarker;
                                adj->searchDist = dist;
              adj->estDist = dist + CalcEstimatedDist(dest->w, dest->l,
                                                 adj->w, adj->l);
                                AddToSearchList(adj);
                                if (adj->nodeType == GOAL)
                                      {
                                      goalFound = TRUE;
                                      searchTrimDist = dist;
                                      MarkPath(adj);
                                      TrimSearchList();
                                      }
                                }
                          }
                    }
              RemoveFromSearchList(cur);
              }
        return(goalFound);
        }


VERTEX_NODE* VertexList::FindVertex(int t, int w, int l)
   {
   VERTEX_NODE* cur;

   for(cur = GetFirstVertex();
      cur != (VERTEX_NODE*)NULL;
      cur = GetNextVertex(cur))
      {
      if ((cur->t == t)
```

```
        && (cur->w == w)
        && (cur->l == l))
           return(cur);          // found it

        if ((cur->t >= t)
        && (cur->w >= w)
        && (cur->l >= l))
           break;                // passed it - it's not in the list
              }
    return((VERTEX_NODE*)NULL);
    }


VERTEX_NODE* VertexList::GetFirstVertex()
    {
    return(vertexHead);
    }


VERTEX_NODE* VertexList::GetGoalVertex()
    {
    VERTEX_NODE*tmp;

    for(tmp = GetFirstVertex();
       tmp != (VERTEX_NODE*)NULL;
       tmp = GetNextVertex(tmp))
       {
             if (tmp->nodeType == GOAL)
          break;
       }
    return(tmp);
    }


VERTEX_NODE* VertexList::GetNextVertex(VERTEX_NODE*cur)
    {
    return(cur->next);
    }


int VertexList::GetSearchMarker()
        {
        return(searchMarker);
        }


VERTEX_NODE* VertexList::GetStartVertex()
        {
        VERTEX_NODE*tmp;

        for(tmp = GetFirstVertex();
                tmp != (VERTEX_NODE*)NULL;
                tmp = GetNextVertex(tmp))
                {
                if (tmp->nodeType == START)
           break;
        }
    return(tmp);
    }


void VertexList::InsertNewVertex(VERTEX_NODE*a)
        {
        VERTEX_NODE *cur;

        if (a == (VERTEX_NODE*)NULL)
                return;

        if (vertexHead == (VERTEX_NODE*)NULL)
                {
                vertexHead = a;
                (*a).prev = (*a).next = (VERTEX_NODE*)NULL;
                return;
                }

        for(cur=vertexHead; cur != (VERTEX_NODE*)NULL;   cur=(*cur).next)
                {
                if ( (a->t > cur->t)
      || ((a->t == cur->t)
        && (a->w > cur->w))
       || ((a->t == cur->t)
         && (a->w == cur->w)
                && (a->l > cur->l)) )
                      {                           // insert after cur
```

```
                if ((*cur).next == (VERTEX_NODE*)NULL)
                        {
                        (*a).next = (*cur).next;    // eol - append new node
                        (*a).prev = cur;
                        (*cur).next = a;
                        break;
                        }
                else
                        continue;                   // get next node
                }
        if ( (a->t < cur->t)
        || ((a->t == cur->t)
                && (a->w < cur->w))
        || ((a->t == cur->t)
                && (a->w == cur->w)
                && (a->l < cur->l)) )
                {                               // insert before cur
                if (cur->prev == (VERTEX_NODE*)NULL)
                        {
                        a->prev = (VERTEX_NODE*)NULL;// at start of list
                        a->next = cur;
                        cur->prev = a;
                        vertexHead = a;
                        break;
                        }
                else
                        {
                        a->prev = cur->prev;        // in middle/end of list
                        a->next = cur;
                        cur->prev = a;
                        cur = a->prev;
                        cur->next = a;
                        break;
                        }
                }
        if ((a->t == cur->t)
        && (a->w == cur->w)
        && (a->l == cur->l))
                {                               // insert after cur at eol
                if ((*cur).next == (VERTEX_NODE*)NULL)
                        {
                        (*a).next = (VERTEX_NODE*)NULL;
                        (*a).prev = cur;
                        (*cur).next = a;
                        break;
                        }
                else
                        continue;
                }
        }
//      AddToSearchList(a);
//      cout << "Inserted vertex (";
// cout << (*a).t << "," << (*a).w << "," << (*a).l << ")\n";
        }


void VertexList::ListAllVertices(int debugFlag)
        {
        char tmp[256];
        fstream debugFile;
        VERTEX_NODE *cur;

        strcpy(tmp, "List of all nodes in list\n");
        cout << tmp;
        if (debugFlag)
                {
                debugFile.open(DEBUG_FILENAME, ios::app);
                debugFile.write(tmp, strlen(tmp));
                debugFile.close();
                }
        for(cur=vertexHead; cur != (VERTEX_NODE*)NULL; cur=cur->next)
                {
                sprintf(tmp, "Vertex:%c: at (%d,%d,%d)\n",
                              (*cur).nodeType, (*cur).t, (*cur).w, (*cur).l);
                cout << tmp;
                if (debugFlag)
                        {
                        debugFile.open(DEBUG_FILENAME, ios::app);
                        debugFile.write(tmp, strlen(tmp));
                        debugFile.close();
                        }
                cur->edgeList->ListAllEdges(debugFlag);
                }
        }
```

```
void VertexList::ListSearchList(int debugFlag)
	{
	char tmp[256];
	fstream debugFile;
	VERTEX_NODE*cur;

	if (searchHead == (VERTEX_NODE*)NULL)
		{
		sprintf(tmp,"Empty SearchList! (marker=%d)\n", searchMarker);
	cout << tmp;
	if (debugFlag)
		{
		debugFile.open(DEBUG_FILENAME, ios::app);
		debugFile.write(tmp, strlen(tmp));
		debugFile.close();
		}
		return;
		}

	sprintf(tmp, "List of nodes in Search list (marker=%d)\n", searchMarker);
    cout << tmp;
	if (debugFlag)
		{
		debugFile.open(DEBUG_FILENAME, ios::app);
		debugFile.write(tmp, strlen(tmp));
		debugFile.close();
		}
	cur = searchHead;
	do
		{
		sprintf(tmp, "SearchList node (%d,%d,%d) Dist = %f",
	      cur->t, cur->w, cur->l, cur->searchDist);
	cout << tmp;
	if (debugFlag)
		{
		debugFile.open(DEBUG_FILENAME, ios::app);
		debugFile.write(tmp, strlen(tmp));
		debugFile.close();
		}
		cur=cur->searchNext;
		}
	while (cur != searchHead);
    strcpy(tmp, "End of SearchList\n");
    cout << tmp;
	if (debugFlag)
		{
		debugFile.open(DEBUG_FILENAME, ios::app);
		debugFile.write(tmp, strlen(tmp));
		debugFile.close();
		}
	}


void VertexList::MarkPath(VERTEX_NODE*v)
	{
	VERTEX_NODE*tmp;

	if (v == (VERTEX_NODE*)NULL)
		return;

//	cout << "Path from Goal to Start\n";
	v->pathTo = (VERTEX_NODE*)NULL;
	do
		{
		tmp = v->pathFrom;
//		cout << "(" << v->t << "," << v->w << "," << v->l << ")\n";
		if (tmp != (VERTEX_NODE*)NULL)
			{
			tmp->pathTo = v;
			v = v->pathFrom;
			}
		}
	while (tmp != (VERTEX_NODE*)NULL);
	}


void VertexList::RemoveFromSearchList(VERTEX_NODE*a)
	{
	VERTEX_NODE *cur;

	if ((a->searchPrev == (VERTEX_NODE*)NULL)
	|| (a->searchNext == (VERTEX_NODE*)NULL))
```

```
	{
	if (searchHead == a)
	   searchHead = (VERTEX_NODE*)NULL;
	return;
	}

	 cur=a->searchPrev;
	 if (cur == a)
		{
	searchHead = (VERTEX_NODE*)NULL;
//		cout << "Removed Search Node (";
//	cout << a->t << "," << a->w << "," << a->l << "). SearchList empty\n";
		}
	 else
		{
		cur->searchNext = a->searchNext;
		cur = a->searchNext;
		cur->searchPrev = a->searchPrev;
		if (searchHead == a)
			searchHead = a->searchNext;
//		cout << "Removed Search Node (";
//	cout << a->t << "," << a->w << "," << a->l << ").\n";
		}
	 a->searchPrev = a->searchNext = (VERTEX_NODE*)NULL;
	 }


void VertexList::RemoveVertex(VERTEX_NODE *todie)
	{
	VERTEX_NODE *tmp;

	if (todie == (VERTEX_NODE*)NULL)
		return;

	RemoveFromSearchList(todie);

	if ( ((*todie).prev == (VERTEX_NODE*)NULL)
	&& ((*todie).next == (VERTEX_NODE*)NULL) )
		{
		vertexHead = (VERTEX_NODE*)NULL;
		return;
		}
	if ( ((*todie).prev != (VERTEX_NODE*)NULL)
	&& ((*todie).next != (VERTEX_NODE*)NULL) )
		{
		tmp = (*todie).prev;
		(*tmp).next = (*todie).next;
		tmp = (*todie).next;
		(*tmp).prev = (*todie).prev;
		return;
		}
	if ( ((*todie).prev == (VERTEX_NODE*)NULL)
	&& ((*todie).next != (VERTEX_NODE*)NULL) )
		{
		vertexHead = tmp = (*todie).next;
		(*tmp).prev = (VERTEX_NODE*)NULL;
		return;
		}
	if ( ((*todie).prev != (VERTEX_NODE*)NULL)
	&& ((*todie).next == (VERTEX_NODE*)NULL) )
		{
		tmp = (*todie).prev;
		(*tmp).next = (VERTEX_NODE*)NULL;
		return;
		}
	}


void VertexList::TrimSearchList()
   {
	VERTEX_NODE *cur, *tmpPtr=(VERTEX_NODE*)0;
	int trimmed=TRUE;

//	cout << "Trimming search list to " << searchTrimDist << " or less\n";
	if ((searchHead == (VERTEX_NODE*)NULL)
    || (searchTrimDist < 0.0))
		return;

    cur = searchHead;
	while ((tmpPtr != cur) || (trimmed == TRUE))
		{
		if (trimmed==TRUE)
			{
			tmpPtr=cur;
```

```
                        trimmed=FALSE;
                        }
                if ((*cur).searchDist >= searchTrimDist)
                        {
//                      cout << "Trimmed out (";
//              cout << cur->t << "," << cur->w << "," << cur->l << ") ";
//              cout << "Dist was " << cur->searchDist << "\n";
                        if (cur == cur->searchNext)
                                {
                RemoveFromSearchList(cur);
                                break;
                                }
                        else
                                {
                                tmpPtr = cur->searchNext;
                                RemoveFromSearchList(cur);
                                cur = tmpPtr;
                                trimmed = TRUE;
                                }
                        }
                else
                        cur = cur->searchNext;
                }
//      cout << "-------\n";
        }
```