Evaluating techniques of finding a path in a dynamic digital search domain

A Dissertation Presented by

Daniel Potter

Letterkenny institute of technology

In fulfilment of part of the examination requirement for the research masters of the Letterkenny Institute of Technology

Supervised by Simon Mc Cabe and Dr Mark Leeney

Submitted to the National Council for Education Awards, February 2009

Declaration

I hereby declare that with effect from the date of which this dissertation is deposited in the Library of Letterkenny Institute of Technology, I permit the Librarian of Letterkenny Institute of Technology to allow the dissertation to be copied in whole or in part without reference to the author on the understanding that such authority applies to the provision of single copies made for study purposes or for inclusion within the stock of another library. This restriction does not apply to the copying or publication of the title or abstract of the dissertation. It is a condition of use of this dissertation that anyone who consults it must recognise the copyright rests with the author, and that no quotation from the dissertation, and no information derived from it, may be published unless the source it properly acknowledged.

Acknowledgements

I would like to acknowledge and extend my heartfelt gratitude to the following persons who have made the completion of this thesis possible.

My supervisor, Simon McCabe, for his time, patience and for being a great source of fresh ideas for the project. I would also like to thank Dr Mark Leeney for his time when the numbers didn't all add up.

Finally my family and friends for keeping me motivated to finish this.

Abstract

Evaluating techniques of finding a path in a dynamic digital search domain by Daniel Potter

Moving from one point to another in an efficient yet effective manner has always been a problem for people. The same problem can be related to mathematical graph theory, where moving from one co-ordinate to another generates similar frustration. The term" pathfinding" is used to describe a process which is purposely designed to solve this problem. Pathfinding is used in many computer applications, not just limited to games, but within games is where its use is most apparent.
Within this context this document initially sets out to examine and critically review pathfinding algorithm.. Special attention is focused upon the A* algorithm, due to it being one the most widely used algorithm to solve the pathfinding problem. These algorithms are analysed in the terms of processing speed and efficiency and from this conclusions are drawn up. It was determined that A* was the best overall algorithm over a set of standard problems that where tested.
The players of digital games have come to expect that the games developed have the latest game technology and programming techniques powering them. With this in mind the dynamic pathfinding process was investigated. Dynamic pathfinding is standard pathfinding, but it is done in a domain where the domain is likely to change of the course of the game. Using similar techniques employed to test the non dynamic algorithms, a dynamic algorithm was analysed and was found out to be very accurate but caused unnecessary processing and memory expense.
In an effort to remedy the discovered problems and add some closure to this research a custom algorithm was developed. The algorithm was tested using the same tests as the other dynamic algorithm and was found to have less processing and memory overheads at the expense of the accuracy of the algorithm.

Table of Contents

Table of figures

Introduction

The following document is the result of research into several key areas regarding computer games and their use of pathfinding algorithms.
Chapter 1 explores what games actually are by considering a definition of games and examining core concepts related to all games. This examination expands into digital games and how they work, paying special attention to artificial intelligence (AI). AI is a very broad term in digital games and is refined by looking at what an AI agent must do in order to be considered functional. This research is interested in an agent's ability to navigate around a digital environment.
Chapter 2 introduces pathfinding and an AI agent to navigate around a digital game environment. The chapter investigates the use of search graphs that represent the game environment. These search graphs are navigated by agents by the use of pathfinding algorithms (PFAs). These PFAs represent an AI agent's ability to navigate an environment and depending upon the algorithm that is used to complete searches, there are varying success levels. Several PFAs are documented, assessed and demonstrated in this chapter. Particular attention is given to the A* (A-star) and D* (D-Star) algorithms, the latter is an example of a dynamic pathfinding algorithm. There is also an examination of dynamic pathfinding and its attempts to solve the dynamic pathfinding problem, as well as how environments can change during the running of a game. Research showed that rather than creating a path, dynamic pathfinding algorithms edit previously generated paths with regard to the change that happened. This chapter also introduces the big O notation algorithm performance methodology.
Chapter 3 documents the development of a test bed application which is designed to visually demonstrate the operation of selected algorithms that have been documented in chapter 2. The chapter details the design of the application from a list of requirements up to the technical implementation of search domain and algorithm requirements.
Chapter 4 takes the test bed application and uses it to demonstrate the effectiveness of the selected algorithms that where developed with it. The tests used were standard for all algorithms, they ranged from simple to complex in order to gain an average value of overall performance for each tested algorithm. To reinforce the data derived, graphs are used to visually display the results of the tests. Special purpose tests for the testing of dynamic algorithms are also devised and used to test any dynamic algorithms.

Chapter 6 is used to develop a solution to some of the problems discovered when researching and testing the documented dynamic algorithm. A solution is derived by analysing problems found and the reengineering of an algorithm already documented to fit the solution. The same tests for dynamic algorithms as demonstrated in chapter 5 are used to assess the performance of the new algorithm and a critical comparison of the results concludes the chapter.

Chapter 7 details the major conclusions derived from each chapter of the document along with suggestions for future research.

1. Digital games

1.1 Computer games

This research has been strongly motivated by, and is inherently linked, to computer games. Computer game systems are repeatedly referred in this document. With this in mind, it would be useful to work from a definition

Zimmermann and Salen have produced one of the most thoughtful texts in computer games and it is their definition that we will use and return to.

"*A game is a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome*"

(Zimmermann & Salen, 2003)

The Zimmermann/Salen quote outlines three core concepts in gaming. Each concept relates to other concepts with varying degrees of dependency in any given game.

- **Rules:**
  Rules in a game are often regarded as the "formal identity" (Zimmermann & Salen, 2003) of the game. Within the Zimmermann and Salen context
    - **Operational:** Rules that directly affect the player
    - **Constitutional:** Rules that affect the game world
    - **Implicit:** General behaviour rules

- **Quantifiable Outcome:**
  The outcome of a game is the measure of player's successes in completing the goals of the game. Typically rules inhibit a player's desire to achieve a goal in the most efficient manner

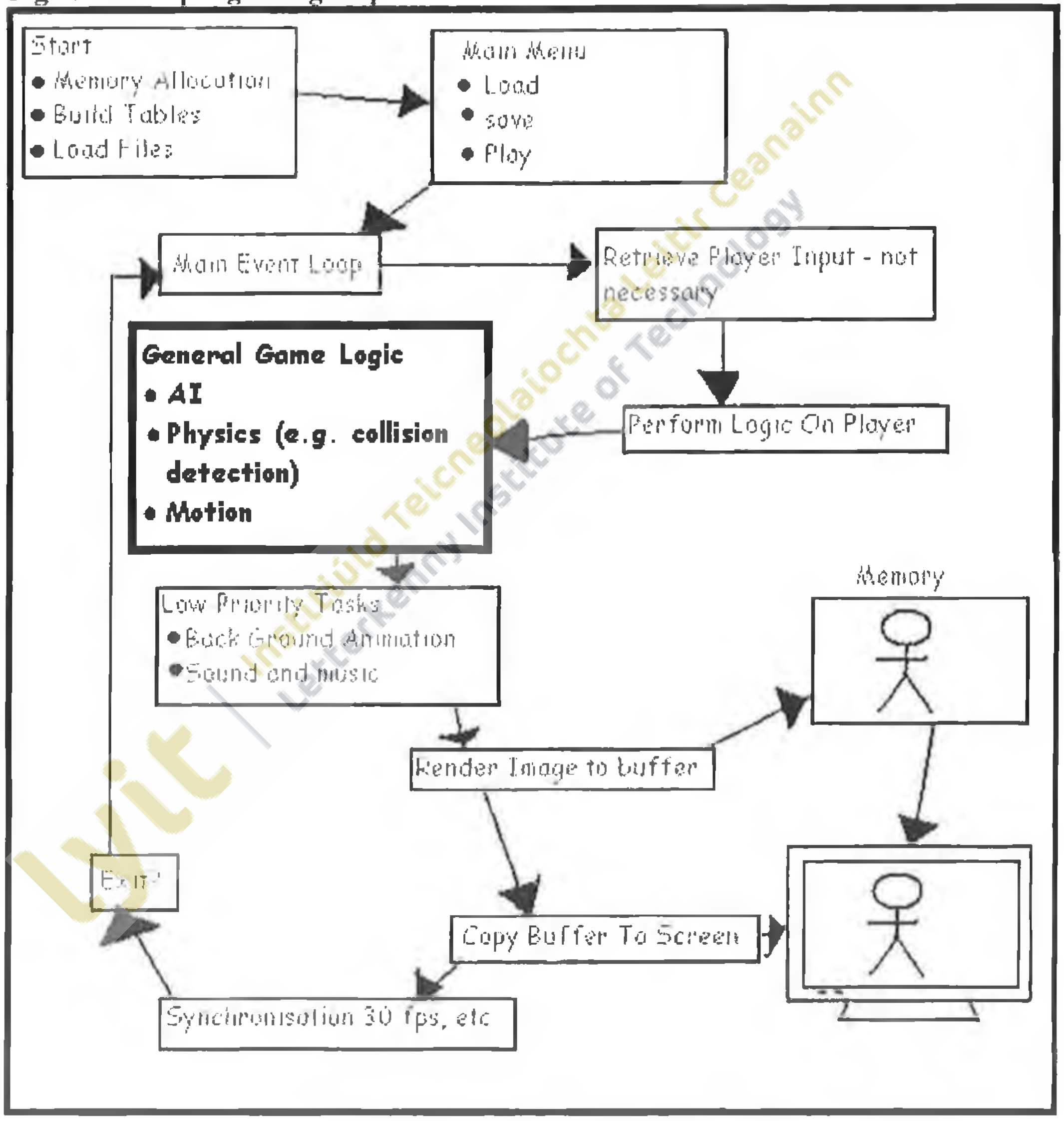
- **Conflict:**
  Conflict defines the contest that must happen in order for the goals to be achieved. This is a key concept to all games, since if there wasn't any conflict in the achievement of the goals of a game, there wouldn't be any point in playing it. Conflict can be provided in a game by other players or by the game rules.

1.2 Computer game high level view

If we move from the more esoteric view of games to one which sits more comfortably in computer science, we must consider the components which make up a modern computer game as a set of programs or loops

The gaming loop constantly iterates during a game, it covers everything from handling game input, to rendering the graphics to the screen. The gaming loop is briefly described below.

Fig 1.1 Example gaming loop



The loop starts by setting up the game. These tasks vary from game to game but a typical example would be preloading game content (graphics, sounds etc...) and

memory allocation. Upon completion, there is usually some kind of menu system to give the player direction into starting the game.
At this point the game loop starts, at each iteration of the loop several tasks are performed.

- Retrieve player input from hardware input devices.
- Perform player logic.
- Perform general game logic.
- Render image displaying current state of the game to the screen.

The loop repeats until a quit condition is activated by the player or the game system. This research is concerned primarily with the game logic performance, since this is where game intelligence will be located and executed.

1.3 Artificial intelligence in a computer game

Artificial Intelligence (AI) is a very broad topic that covers many areas of computing. It has been applied to image recognition, expert systems, weather prediction etc. As such the definition of AI within each of these areas varies greatly. To conceive a definition of AI within games, we need to examine what exactly AI does within the game.

It can be seen from Fig 1.1 that AI plays a key role in the game loop and from the book *Game design theory and practice* (Rouse, 2001) it is stated that one of the goals of AI in a digital game, is to challenge the player. Typically AI is used in relation to the inhibiting factors in a game that deny the player a victory state. In actual performance this may simply mean 'bad guys' within a game (people, machines, ships etc... that attack or distract a player)

1.4 A real time AI agent

We will consider 'bad guys' etc, as 'AI agents'. An AI agent is a distinct entity within the game with its own AI logic, which in turn performs within the AI logic of the game.

An agent could be viewed as a separate entity from the game although it performs on behalf of the game. Using the systems mentioned later in this section, it can be made aware of any game event then it can deduce what to do from its decision making system.

It is common to also give an AI agent local priority decision making and frequency of execution. The higher the priority the more often the agent will perform in the gaming loop.

AI agents have designated goals which they must complete; these goals usually are set to inhibit the successful completion of challenges by the player. For example, in an adventure game the player is trying to claim the treasure, while any AI agent goal could be set to try and stop anyone from claiming the same treasure.

For an AI agent to work effectively in the gaming environment it needs to be able to perform three major functions outlined by Ed Byrne in his book *Game Level Design* (Byrne, 2005), Byrne states that AI requires processing in the following ways.

- **Awareness:**
  An agent's awareness refers to the game environment and the current state of play e.g. agents location, behaviour of other agents, weapon held by player etc. The scope of environmental variables that an agent is exposed to is defined by the game designer. The parameters will have an obvious impact on logic processing and general AI complexity.

- **Decision Making:**
  Decision making is the ability of an AI agent to make a decision on what to do in the game depending upon environment variables. A decision by an agent could also be affected by its class as an agent, for example one agent might be classed as "intelligent" while another might not.

- **Navigation:**
  Navigation regards how an agent traverses the gaming environment. This includes both avoidance and pursuit of environment objects (e.g. player, doors etc). Navigation in modern games is achieved through the use of pathfinding.

Pathfinding denotes the use of a graph representation of the game environment; these graphs are discussed in detail in the following section. The graph is used by the AI agent to navigate around obstacles and move to a different area of the environment. Traversal of graphs is determined by several algorithms that owe their origin to graph theory. These are called Pathfinding Algorithms (PFAs). The traversal of the graph depends on the PFA employed in the task. The debate about the best route for a given graph is one that dogs PFAs, since "best" is often a subjective issue. In the next chapter we shall consider PFAs in more detail, as well as the items affecting them.

1.5 References

- Steven M. Woodcock, http://www.gameai.com/ ,accessed 2/10/05, Last Update 7/10/05.
- Rules of Play, Eric Zimmermann and Katie Salen, October 2003, ISBN 1556227353, the MIT press.
- Game Design Theory and Practice, Richard Rouse, February 2001, ISBN 1556227353, Wordware Publishing Inc.
- Game Level Design, Ed Byrne, 2005, ISBN 1-58450-369-6, Charles River Media.
- Game Architecture and Design, Andrew Rollings & Dave Morris, 2000, ISBN 1-57610-425-7, the Coriolis Group.
- Introduction to 3D Game Engine Design Using DirectX 9 and C#, Lynn T. Harrison, 2003, ISBN 1-59059-081-3, Apress.
- Windows Game Programming for Dummies 1st Edition, Andre LeMothe, 1998, ISBN 0764503375, Hungry Minds.

2. Pathfinding in computer games

2.1 Navigation in a game environment

The previous chapter gave a brief description of how an AI agent navigates around an environment in a digital game using pathfinding. This chapter examines the issue of pathfinding and how we can relate this problem to any environment be it digital or real.

2.2 The pathfinding problem

"*Looking for a good route for moving an entity from here to there*"

(Stout, 1997)

This is a somewhat brief definition, but one which introduces some of the issues with pathfinding. The definition introduces the idea of "*looking*", possibly indicating that pathfinding considers something, namely a "*good route*". The route can be viewed as the route that the entity must take to get from "*here to there*". But what is a good route? What is a bad route? These questions are important to the research of pathfinding algorithms.

"*Pathfinding is simply the process of moving the position of a game character from its initial position to a desired location*"

(Bourg, 2004)

In Bourg's definition, the core idea is the process of moving from one point to another. It implies nothing of the process of searching or 'looking'. It can be considered as a goal of pathfinding rather than a description of it.

The quotes are presented as an illustration of some of the ambiguities that exist in pathfinding. If agreement can't be sought in the definition of the problem, how does one present a solution? It is the goal of this research to side step some of the problems by considering pathfinding only in a gaming context and with a strict definition.

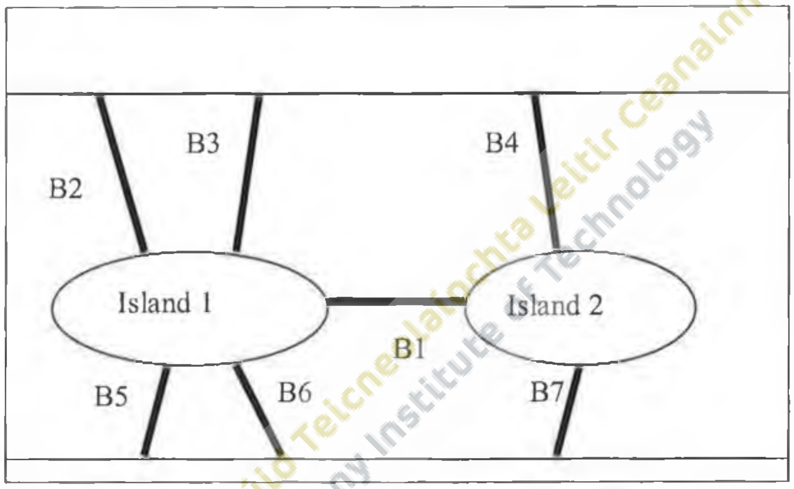
2.3 Origins of pathfinding

To better explain how pathfinding works it is useful to see where its origins lie. The catalyst to the modern pathfinding debate was the "Bridges of Königsberg problem" proposed by L. Euler.

Königsberg is located upon two islands in the middle of a river, its location and physical properties made it ideal as a problem space for pathfinding. Euler's proposition was to find a route around the city that encompassed all of the bridges without using the same bridge more than once, networking from the start point.

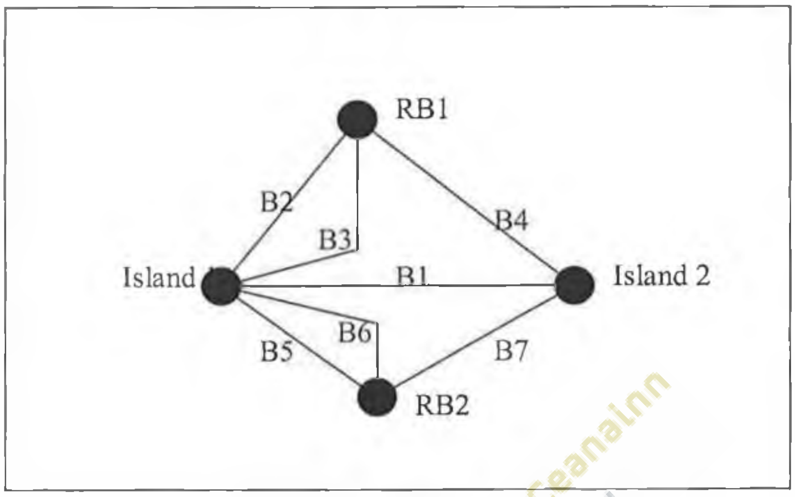
The layout of the city and its bridges was similar to that shown in **Fig 2.1**.

Fig 2.1 Bridge of Königsberg problem



Euler proposed looking at the problem at its logical level articulated in a graph. This allowed him to consider the problem without extraneous distraction; it was the start of graph theory. The graph that was created of Königsberg would have looked something like **Fig 2.2**.

Fig 2.2 Graph of bridge of Konigsberg



"Graph preserves essential structure of the bridge system, while ignoring extraneous features such as distance or direction"

(Luger, 2001)

The Luger quote is included to highlight that the graph in **Fig 2.2** is an exact representation of the bridge layout in **Fig 2.1**. The graph however makes the problem easier to solve as it ignores features of the bridge layout which could possibly make the problem harder to solve. More importantly than this however, is that in Fig 2.2 we have a graph which we can use in computing.

2.4 Modern use of conceptual maps in game

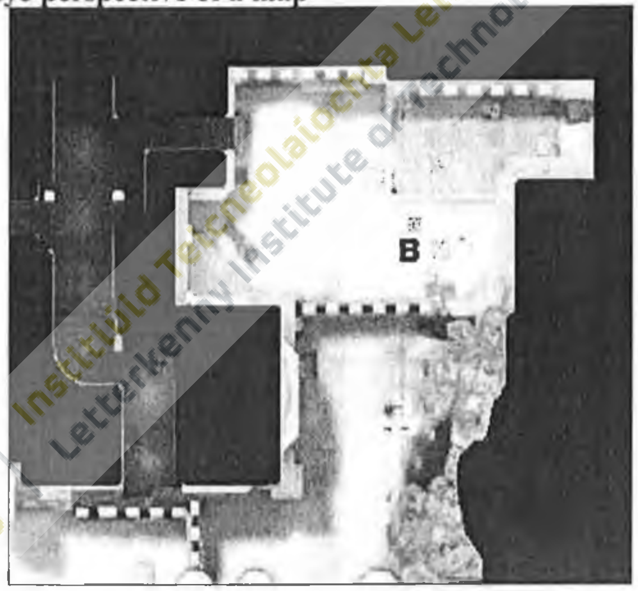
Whereas Euler took the real world then mapped it to a virtual space, games are virtual spaces with real world elements mapped onto them. Games are inherently related to graph theory.

Game environments are built using tools or middleware such as the Valve hammer level editor (Valve hammer level editor, Accessed 14/12/07). While environments can be built in much the same way as building a house out of blocks, the level designer must always be aware of the graph that they are creating, as this will have a heavy impact on the pathfinding process of their AI agents.

Consider the maps screen shots taken from the map de_dust2 from the popular online game Counter Strike Source (Counter Strike Source Official Website, Accessed 3/3/06).

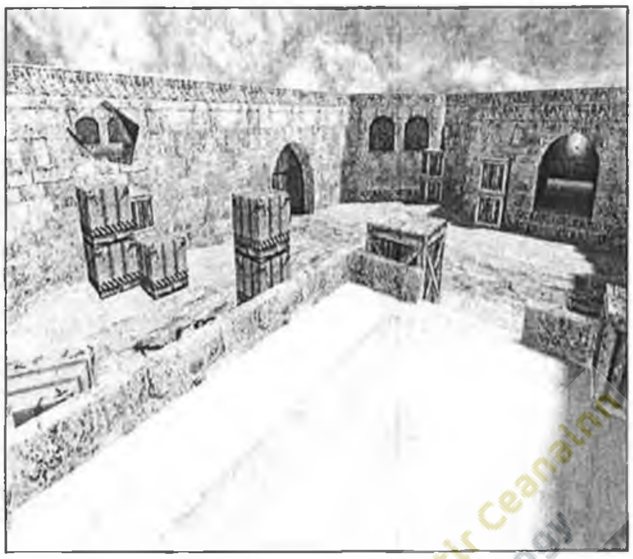
- 2.3 shows a map of a section of the level.
- 2.4 is the visual aesthetic of the same section
- 2.5 shows the map with nodes and edges
- 2.6 strips the visual detail to give us a conceptual map of the section

Fig 2.3 A bird's eye perspective of a map



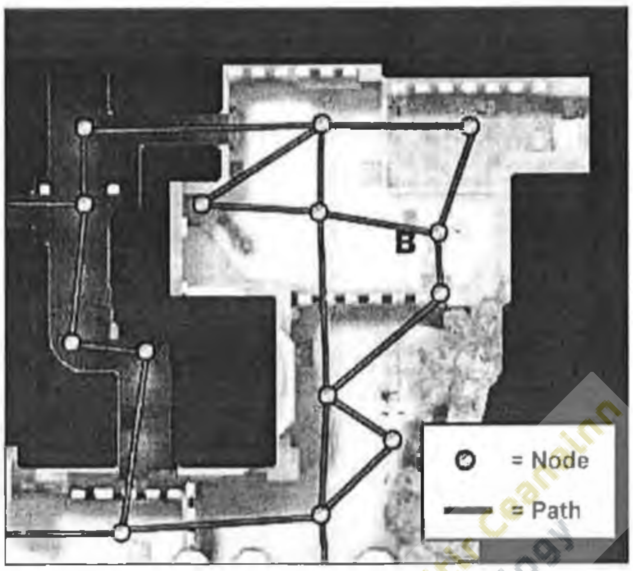
The level designer needs to be able to recognise features in the map that would affect the agent AI such as the doors, corners and walls.

Fig 2.4 A fixed in game camera viewpoint of the same section of map



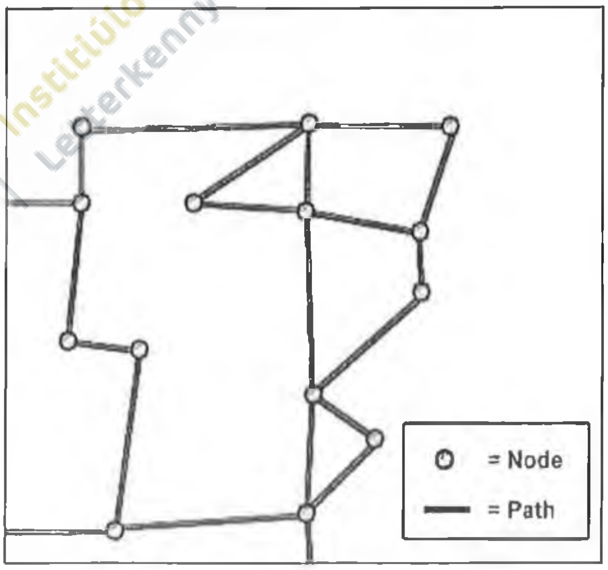
Depending on the kind of game being created the map could also tell the agent something about the area that it is in. It would be common practice for level designers to embed information into a map so that an AI agent can use it to make accurate decisions. For example, if a node was in a dark corner of a room, the information can be embedded into that node informing it that it is dark, and from that information the AI agent's decision making could possibly make it hide there.

Fig 2.5 Birds eye view with nodes and edges



The map created would be more complicated than what it shown in **fig 2.5**. The diagram shows the nodes and the paths that the agent would use to navigate the area.

Fig 2.6 Nodes and edges, our conceptual map of the area



Finally if the map is removed we have a graph similar to the one Euler designed for the bridges of Königsberg problem. It has all the information necessary for the traversing of the game environment while not having distractions such as lights, shadows and noises.

2.5 Pathfinding algorithms

Pathfinding algorithms are essentially search algorithms that can be used to search the graphs of a gaming environment. There are many algorithms available for the task, each with their own strengths and weaknesses and different methodologies for searching graphs.

Simply, algorithms can be grouped into "one step at a time" and "look before you leap" algorithms. Informed or Uninformed algorithms are a subset of the "look before you leap" methodology and will be explained later.

2.6 Algorithm performance

One of the most difficult tasks in developing a game is to try and make it as realistic as possible e.g. with good AI agents, these processes cam have a heavy effect in processing and will potentially slow down or stall the visual output of a game. It is therefore the goal of PFAs not only to search graphs realistically and speedily, but also efficiently in terms of processing power.

In relation to the performance of an AI agent, two factors are key. The accuracy of searching and the speed of the search taking place.

- **Accuracy:** This reflects an agent's ability to find the shortest path between two points on a graph using a pathfinding algorithm. The shorter the path the more accurate it is and hence the more the agent will be perceived as being 'intelligent' by the player.
- **Speed:** The creation of the path between two points requires the use of a pathfinding algorithm. Speed is relative to how fast the algorithm can create a path between the two points. The shorter the processing, the less affect it will have on the overall performance of the game

Both speed and accuracy are additionally hampered by factors outside of the algorithm itself and beyond its control such as large search graphs. For the purposes of this research we will have to ignore them and focus on relatively small graphs. Before we can begin to test the speed and accuracy of each pathfinding algorithm, we must first make reference to an objective system for measuring algorithm performance.

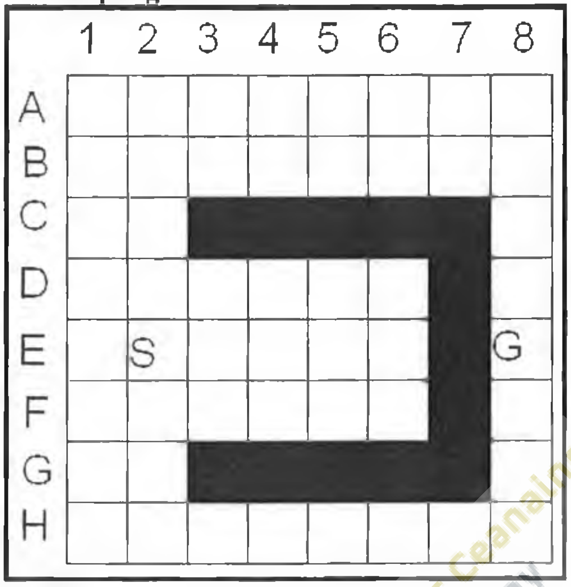
2.7 Big O notation

Big O notation is perhaps the most common method used in the comparison of algorithms. The "big O of an algorithm" is a function which measures how much work is done by an algorithm to process data sets of various sizes. This allows analytic comparison of different pathfinding algorithms in different search domain. It will also allow a statement of some objective truths about PFAs and their operation in comparison to each other.

2.8 One step at a time

One step at a time refers to a category of PFAs which share similar characteristics in their performance. They are sometimes referred to as blind search algorithms instead of searching a graph from a start point as what happens with other algorithms. The AI agent will itself position itself at the next node of each search, then making a decision which node to move to next. An analogue would be traversing a country without a map, making decisions upon where to go next based upon signposts encountered on the route.

Fig 2.7 Sample grid



The grid in **Fig 2.7** shows a sample grid. Any open grid space is a viable move for an algorithm while any space that is black represents an obstacle in the game world. S is the start and the G is the goal of the algorithm. The grid layout is not dissimilar from the graphs shown in section 2.4, since any of the open spaces would reflect a node in a graph and where the squares are joined would be connection between the nodes. The obstacles would not be included since they are what being worked around in the graphs above.

One step at a time algorithms in general work by exploring the domain until the current location for the search collides with an obstacle, and then it performs the algorithm specifics. In the following sections the following examples of one step at a time algorithms are examined.

- Breseham line algorithm
- Random bounce
- Wall tracing
- Breadcrumb pathfinding

2.9 Breseham line algorithm

This algorithm works on the "line of sight" principle, if the agent using the algorithm can directly see the goal that it wants to move to it will move to it in a straight line. The algorithm doesn't know how to deal with obstacles in the environment so its usefulness is limited to obstacle free gaming environments which realistically don't exist too often. Its inclusion in game is necessary since it is a part of an algorithm that we will consider later (section 2.10 & section 2.11).

- **Speed:** The algorithm works extremely fast since it constantly moves in a straight line to the goal.
- **Accuracy:** Since it only works effectively in obstacle free environments, it would suffer in any modern game and would probably fail to reach the destination goal.

2.10 Random object avoidance

"*Ignore obstacles until one bumps into them*"

(Stout, 1997)

This consists of several algorithms, which to begin with, all use the Breseham method discussed in section 2.9 to move towards the goal in a straight line, if it hits an obstacle during its movement it tries to navigate around the obstacle.

This particular algorithm is sometimes called Randombounce. With this particular method, once an obstacle has been hit in the environment using Breseham, it changes its direction randomly with the hope being that the random direction change will enable it to traverse the obstacle. More intelligent versions of this method try to change the direction so that the search isn't being taken away from the goal. In other words, it tries to prevent backtracking within the search. It will always try to change the direction so that it will move towards the goal.

This kind of algorithm was used in early computer games since they had low performance requirements and simple game environments.

The following is an example of the random direction algorithm.

Fig 2.8 Random direction sample

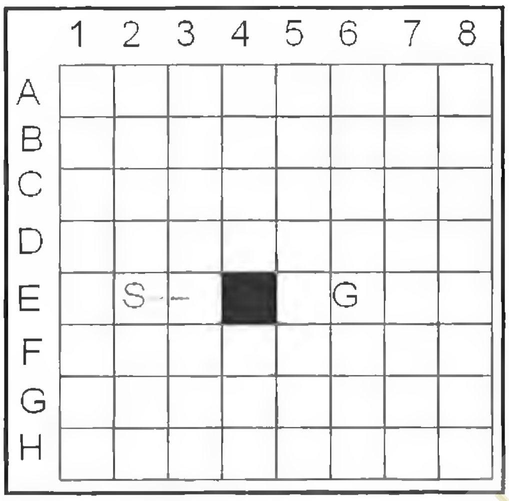
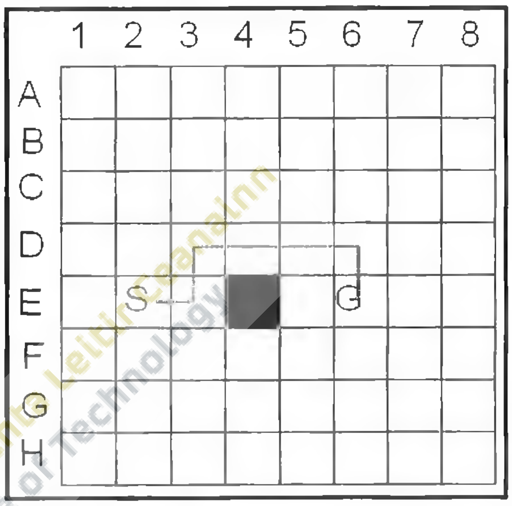


Fig 2.9 Random direction sample



The start node is in (E,2) and the goal node is in (E,6), using Breseham the agent follows the line of sight and it stops in (E,3) when it realises that it cannot move due to the object in (E,4). The agent employs the random direction change. It doesn't want to move back because that is where it came from and at this point there is no benefit in moving either left or right. The agent moves left, this could be determined randomly or by a predefined sequence. The agent then moves forward to space (D,3) and changes its direction to try and move it towards its goal again (turns right). The agent sees that its intended next space is empty so then it moves to it, it continues along with this until it reaches the goal in **Fig 2.9**.

Because the process stated above is very economic for a processor and the algorithm is relatively simple to follow, it makes it seem like this is an ideal solution for pathfinding, unfortunately it isn't. This algorithm is really only effective in situations that are similar to the above, if the environment is complex (which most gaming environments are) then the amount of time it could take the agent to reach the goal could grow enormously and there is a possibility that the agent could get stuck totally. Look at the following situations.

Fig 2.10 Random Direction Problem

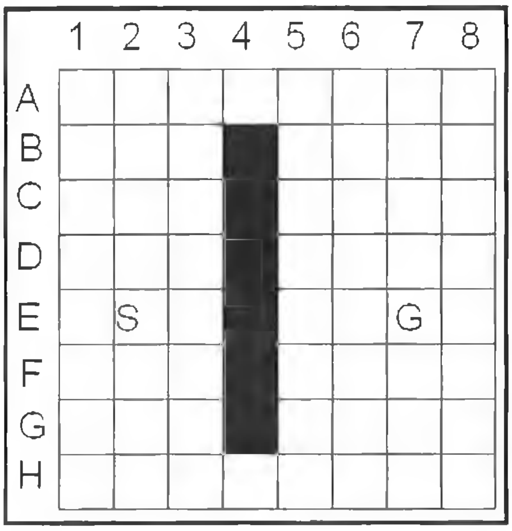
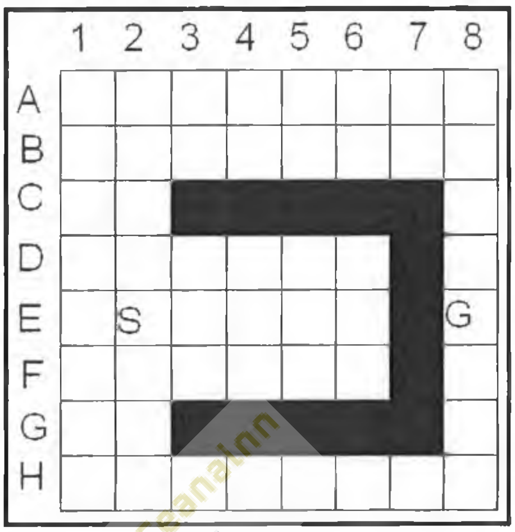


Fig 2.11 Random Direction Problem



In **Fig 2.10** the agent is presented with a long vertical obstacle, it would get to the object and try to traverse it. But because of the nature of the algorithm and its random movement the agent will more than likely keep scaling up down the object trying to find a way past but not actually succeeding. Eventually it will take an extra step and manage to actually traverse the obstacle. But it is less efficient than other algorithms.
In **Fig 2.11** the shape of the object that the agent has to traverse is concave in shape; the agent using the algorithm simply cannot get out of it once it has entered it. It tries to traverse out, but will always try and get back to the goal node, thus moving it back into the obstacle.

- **Speed:** The algorithm is very fast due to its simplicity.
- **Accuracy:** The algorithm can find a path through simple environments effectively but will not guarantee finding the best path available. It has major problems when faced with certain types of obstacles or environments that are highly populated with obstacles. Even with the more intelligent direction selection it is not guaranteed to find a path to the goal, let alone find the shortest path.
- **Usage:** Only really useful in game environments that are grid based and are sparsely populated with obstacles.

2.11 Tracing around obstacles (wall tracing)

This analogue to trying to find a way around an obstacle in the dark, meaning that if this was a human doing the pathfinding they would place a hand upon the obstacle to be traversed. Then gradually work their way around the obstacle until they reach the point at which they wish to stop traversing, it is determining when this point has been achieved that is the main problem with this algorithm.

Two methods for achieving this are to use a variation of the Breseham line of sight algorithm. The first involves traversing the obstacle until a direct line of sight has been found for the goal. The main problem with this is that it relies on being able to see the goal node which is not always possible, especially when it is considered how complex modern game environments are.

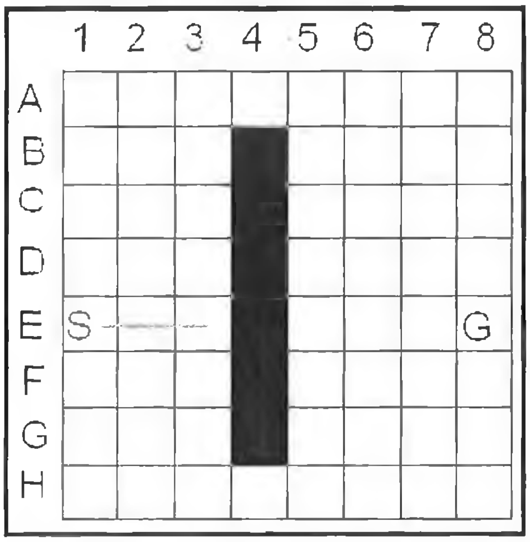
The second is to calculate the equation of the line between the start and goal and then using this to stop the agent traversing the obstacle once it reaches the line. Once it stops the traversal the original behaviour would recommence and the agent would start moving towards the goal.

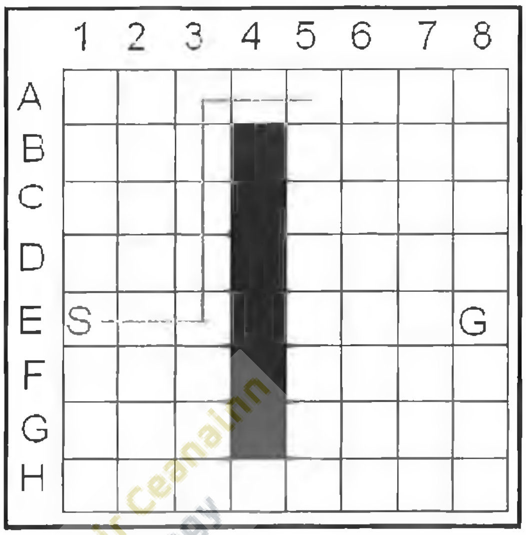
There is another less reliable method which involves cancelling the trace when the agent returns to the original direction that it was going. Again this is totally unreliable and will work only in certain situations.

These methods are only suited to grid based game environments since their workings depend upon the ability to detect obstacles in the path. Since graphs don't store a memory of obstacles, they are not of much use.

The following is a demonstration of an effective execution of the wall tracing algorithm on a problem.

Fig 2.12 Wall Tracing **Fig 2.13 Wall tracing**





In **Fig 2.12** we have the object that gave us a problem with the random direction algorithm. The agent starts at (E,1) and moves along the line of sight until it hits the obstacle in position (E,4). At this point depending upon whether the agent is using left or right wall tracing, the agent will change its direction. If the agent is designed well then it can decide the better direction to take in order to get it closer to the goal, but this may not always lead to the shortest route. This is a major problem of an exploratory algorithms.

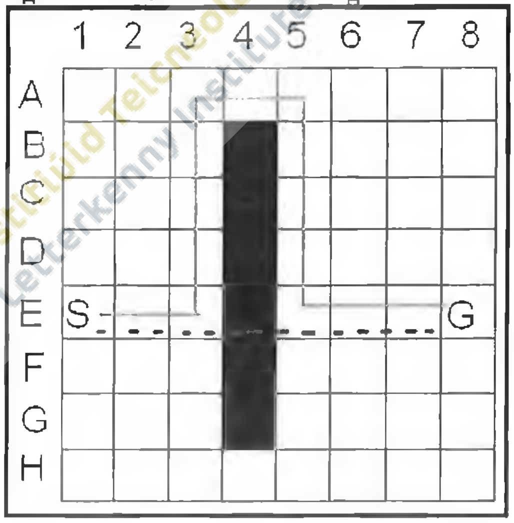
The agent in this example is using right wall tracing meaning that it will try to run a virtual right hand along the obstacle. This results in a turn to the left and continues to trace around the obstacle turning keeping the obstacle on its right. At this point the agent is doing a trace and as mentioned earlier, it is necessary to execute a stop or the agent will simply constantly traverse the obstacle.

Again there are several ways to execute a stop. The simplest method is when the agent is facing in the original direction that it started (e.g. above east) it will stop wall tracing and move back into the line of sight algorithm. In **Fig 2.13** at point (A, 4) it will stop wall tracing using this method.

The second method uses the variation of the Breseham line of sight algorithm to find out if the goal can be seen from the current position. If the agent can see the goal then it will simply stop wall tracing and move towards the goal. In **Fig 2.13** the agent would stop at node (A, 4). This is more expensive than the first method since it requires further computation to see if the goal node can be "seen" but it will make sure that the agent stops in a correct place i.e. if it can see the goal node. However, if during the traversal of an obstacle, the goal node would never becomes "visible", the agent would never get to it destination at that point.

The final method guarantees that the agent will find its way to the goal, but it is more expensive than both previous methods, it is called *robust tracking*. Before the agent begins, an equation of the line between the start node and the goal node is calculated, if while tracing that line is crossed then quit out of tracing a path and begin to move towards the goal again.

Fig 2.14 Robust wall tracing



In **Fig 2.14** the agent traces around the obstacle and knows it has to stop when it hits the line (shown in red dots). It then moves towards the goal.

There are more calculations required for this method than either of the previous methods since the equation of the line is required and at each step the algorithm needs to be able to check to see if it has met the line.

This algorithm will find a path to the goal node, however it is highly unlikely that it will find the optimal route and can often take much more time than the more complex algorithms

- **Speed:** Again like all these exploratory algorithms it runs very fast. The only real calculations that take place are the line of sight and equation of the line methods.
- **Accuracy:** Very poor, the first 2 methods cannot guarantee finding the goal at all, while the third method will find it but it could take an unnecessary amount of time. And hence it could run a very long path.
- **Usage:** This algorithm main usage is limited to one particular type of game, namely maze-based games. An obvious example is Pac-Man. It is still used in some modern games to simulate "dumb" agents.
- **Possible Upgrade:** In research undertaken, there was no mention of a simple solution to stopping a constant repetition of traversing which would be when the agent reaches the point at which it began its traversing. The agent would know that it has completed a full circuit of the obstacle and needs to try another method of pathfinding. Or if the agent is heading in the direction of which it started, reverse the left or right pathfinding to the opposite to try and find a way around.

2.12 Breadcrumb pathfinding / path following

This is the final exploratory method considered in this document. Its main goal is to follow a predefined path in a level in game environment. One could argue that it isn't a PFA at all. It might be called exploratory, given that when the agent is following the predefined path, it could be exploring neighbouring nodes for game elements such as health, enemies.

All the PFA does is check its neighbouring nodes to see if there is a path available to it, then simply moves to it and repeats the process. It needs to be able to remember which node it came from so that it doesn't back track upon itself.

This type of exploration is used in both graph and grid based environments, since it doesn't need to know where obstacles are in the environment, it just needs to know where the path is.

The path would be created by the level designer.

Fig 2.15 Path Following

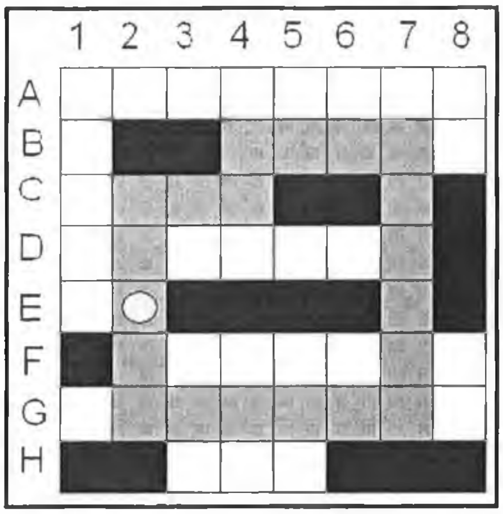
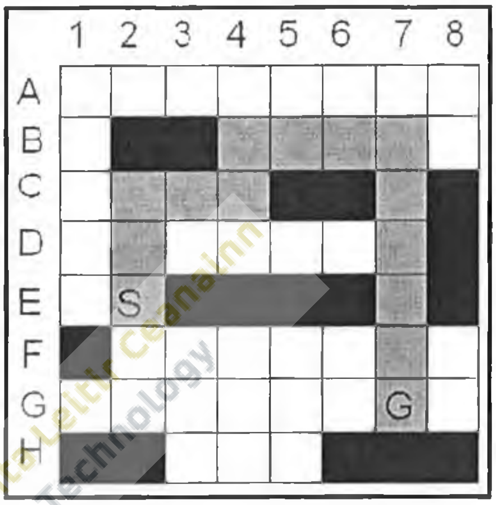


Fig 2.16 Path Following



In **Fig 2.15** the dark grey areas are the path that the agent (located at (E, 2)) must follow. What the agent does when it begins to follow the path is to analyse all of the nodes that surround it in the same way that it looks for an obstacle. It is looking for a path node that it can move to instead. The agent doesn't have to worry about avoiding obstacles as the design of the path itself should guide it around the obstacles. There is no start or goal node in **Fig 2.15** has no start and end node, it is just a constantly repeating path.

Fig 2.16 has a start and goal node associated with it. This allows agents to move over and back from the start to the goal. The agent would follow the path until it reaches the goal. The goal node then itself becomes the new start node and the old start node becomes the new goal node. This would allow the agent to retrace its steps exactly. This technique is often used to simulate a character patrolling a predefined area. If at any point the agent has to leave the path it needs to store the node at which it diverted from the path to perform some game specific situation (e.g. investigating a noise). This is so it can return to the path and continue its patrol if the agent makes it back to the path.

Breadcrumb pathfinding takes it name from the Hansel and Gretel story where the siblings tried to find their way out of a forest by leaving a trail of breadcrumbs behind them. The same applies within a game environment. A certain element within a game can be nominated to leave the trail of breadcrumbs behind it. Thus, creating a path for agents within the game to follow. A good example would be a game character walking across snow leaving footprints. These footprints would then be viewable by other agents and the agents would be able to use the path following methods stated above to follow the footprints to try and find the goal.

- **Speed:** Again this algorithm runs extremely fast since there is very little actually going on with it
- **Accuracy:** Since it is following a predefined path created within the game, it cannot fail to be accurate. If it wasn't accurate the blame would lie with the game designer.
- **Usage:** This method is relied upon in racing games since the predefined path can represent the course or track that is being traversed. Different nodes can represent different pieces of the course. Likewise it can also be used to simulate movement around cities and other environments where roads and walkways can be defined as predefined paths for cars or pedestrians to travel on.

It is also very useful to simulate the patrol areas of guards or soldiers in shooting and stealth games. It cheaply (processing wise) gives realistic looking patrol areas for soldiers.
All these methods are still used in modern games, since they are very effective when used in the correct situation.
In increasingly large, open, explorable games we cannot pre plan every path for an AI agent. The following section aims to address this issue by examining algorithms that create these paths at run time.

2.13 Look Before You Leap

To this point, all the algorithms considered have been exploratory algorithms. But, the effectiveness of these algorithms in modern gaming is questionable.

This section introduces PFAs that are better suited to the graphs generated by modern games. If a path is found, then it will be transformed so that the path following algorithm mentioned in section 2.12, can simply follow it a step at a time.
The algorithms that will be mentioned within this section are search algorithms; they have to search the gaming environment for the best route that the algorithm being used can muster.

2.14 Search Algorithms

"*A graph search (or graph traversal) algorithm is just an algorithm to systematically go through all the nodes in a graph, often with the goal of finding a particular node*"
(Cawsey, 2005)

The above quote is useful in describing what search algorithms are and what they do. The problem that is described is to find the shortest possible route allowable by the algorithm, between a start point and an end point. The term allowable is used since the accuracy of the different algorithms varies. The input is a graph, very much like the one in **Fig 2.6**, with a start position and a goal position (it is possible to do searching with multiple start and goal positions but this area doesn't apply to this research).

Fig 2.17 Graph of nodes

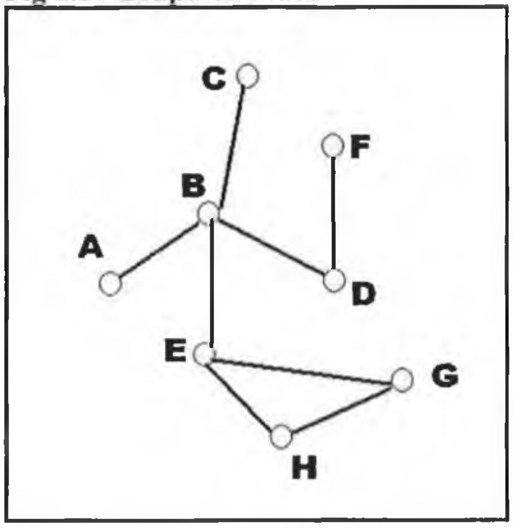


Fig 2.18 Tree representation of graph

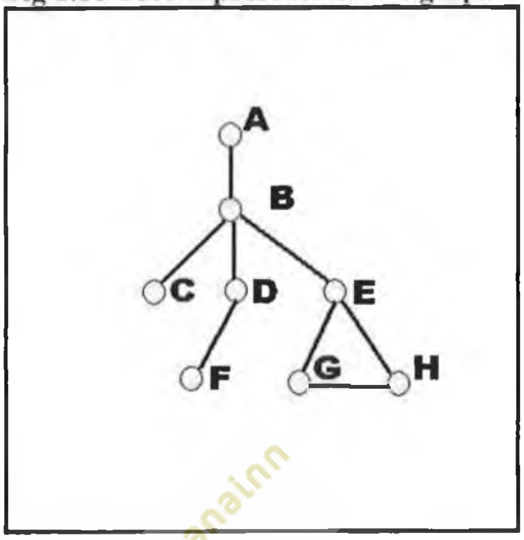


Fig 2.17 shows a graph consisting of nodes from letters A – H, connected to one another by connectors. It may look like a bit complicated as it is, but it can be relatively easily shown as a tree diagram if the start node for the search is node A, as shown in **Fig 2.18.** The tree diagram is much simple to understand and traverse, it is also relatively easy to convert to a computerised structure, to use since some data storage methods are quite similar in the way that the above tree is formed (e.g. linked lists and binary trees).

Linked lists are a data storage technique that can be used to store a graph in computer memory and at the same time allow for easy traversing. A linked list must have a start (not necessarily the start in the search), possibly the first node that is loaded into memory. From this start we can create links (memory addresses) to other nodes that the original start node is joined to. For example if we had the node B from **Fig 2.18** we would need to create links to nodes C,D and E. This is done for the entire graph and once the entire linked list is loaded up into memory, the agents can work on it with their algorithms to search it.

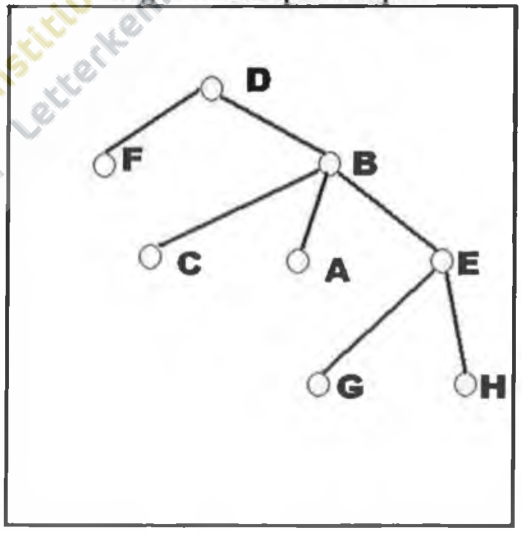
Only the graph for the level/ environment that is currently in use is loaded up into the faster R.A.M of a computer. Graphs for other levels/environments are stored on a slower more capacious media which can be accessed when a level changes (CD-ROM, hard drive).

2.15 Uninformed Algorithms

Uninformed algorithms know nothing of the domain they are operating in. They just perform their tasks on the search space. This is a good characteristic since they generally run very fast since they do not require information about the search environment making it fast, but it can also be a negative characteristic since they are unable make decisions based on the gaming environment, making it inaccurate because it doesn't find the shortest and hence most accurate path.

The best example of an uninformed search is the linear search where items are searched one at a time until the desired one is found. For example in a name database a search was carried out for the name Sarah, the search would search all the names beginning with A then B etc until it finally reached S and find Sarah. All the time spent searching the letters before S is misspent. The search was uninformed and didn't know anything about the alphabetical listing of names.

Fig 2.19 Sample Graph



A more games relevant example is that shown in graph in **Fig 2.19**. If we wanted to find node E, the algorithm would probably search the branch with Node F on it first. Once it got to branch B it would find Node E. This seems fine, but if node F had many branches stemming off it, efficiency would drop considerably.
In general uninformed algorithms are speedy due to their simplicity. But the simplicity may lead to inefficiency and the unnecessary usage of both time and resources.

2.16 Informed Algorithms

Informed algorithms "know" something about the domain that they are in and hence are more "intelligent" than the uninformed algorithms. What "they" know enables them to move more accurately through the domain. Using the name database example with the name Sarah; the informed algorithm could know that it is searching letters, so therefore may only search the first letter allowing it to only search the 18 letters before it (if it is the letter "S"). Once it has located the names beginning with S within the list. It can then begin to search within the names for the name Sarah. The search domain is simplified because the algorithm had useful information about the search domain, allowing for a more complex but far more effective and efficient search.

Within pathfinding, informed algorithms use a heuristic to try and make the search more efficient by trying to estimate the distance between where the search is at the current moment and the goal node. Algorithms that use this particular method search nodes that are closest to the goal to try and shorten the search.
Again using **Fig 2.19** and using the same example (looking for node E), the algorithm would use the heuristic and see that the branch with Node B on it is closer to any other branch to Node E. It would search this node now. This saves time searching any branches that would have been on Node F.

Informed algorithms require more processor time from the computer since they require much additional computation, especially for the heuristic.

2.17 Search algorithm terminology

The proceeding algorithms will all be programmed and tested, therefore:
Before they are investigated, it is useful to become familiar with some of the memory structures used within them.

The first data structure is a queue; it uses the F.I.F.O (first in first out) method.

Items are "pushed" onto the queue and as each item is "pushed" on the items before it are pushed further up the queue. When an item is removed from the queue (in the case of pathfinding, a node would be removed so that it can be searched further), the item at the top of the queue is removed or "popped". The item that was after it then becomes the front item in the queue.

The next is a stack; it works in a very similar way to the queue except that it uses the L.I.F.O (last in first out) method, meaning that the most recent push to the stack is the first item to be popped off. This is normally the item at the bottom of the stack.

There is another version of the queue called the priority queue which is similar to the queue in structure. However this data structure requires additional processing whenever there is an item pushed onto it, and it isn't simply placed at the bottom of the queue. It is placed with some particular rank associated to it.

Fig 2.20 Priority queue example

Priority Queue
Rank 1
Rank 3
Rank 4

In **Fig 2.20** we have a priority queue with three unnamed items on it, each with its own rank. The queue is ordered by the lowest rank being at the top therefore Rank 1 is at the top.

If we tried to push a new item with a rank of two into the priority queue, it would not be placed at the bottom as with the queue and stack. There would need to be additional functionality to place the item in the correct position, the addition would need to search through the queue until it found the correct position. In the example the correct position for the rank 2 item would be in just below the rank 1 and above the rank 3 items. If an item with the same rank as one that already exists within the queue is "pushed" onto it. When an item is "popped", it is the item at the top of the queue that gets removed.

2.18 Breadth First Search

This is an example of an uninformed search algorithm mentioned earlier, it knows nothing of its search domain and the way that it works reflects this. Its internal workings could be compared to the way water flows across a floor. The source of the water being the start node, it spreads evenly in all directions, circumventing any obstacle that it might hit simply by flowing around it and eventually it will reach the goal node. The larger the area that the water covers, the more water is needed to make the water flow further. Therefore as long as there is a path to the goal node this algorithm will find it. It might not find the best path though.

From the start node it searches all the neighbouring nodes (nodes that the original node is connected to). Each node as it is searched is placed onto a queue (data structure mentioned earlier) this carries on until there are no more nodes left to search of that particular node. The first node in the queue is then "popped" off and is searched with the same method as before. This is repeated for all elements in the queue, nodes that have been searched are flagged as searched to prevent back tracking. As you can expect the queue grows in size dramatically over time and the larger the queue the potentially larger the search time. The search ends whenever it finds the goal node or it runs out of nodes to search.

Fig 2.21 Breadth First Searching

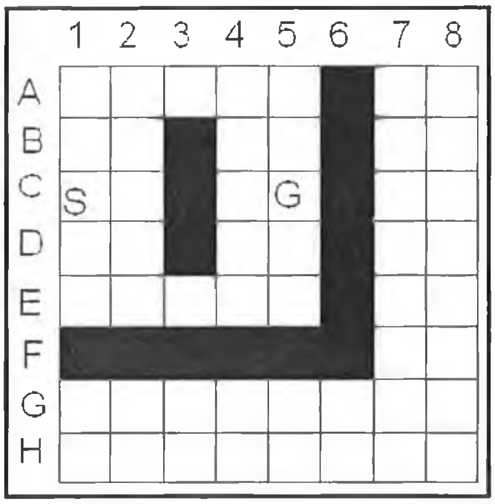
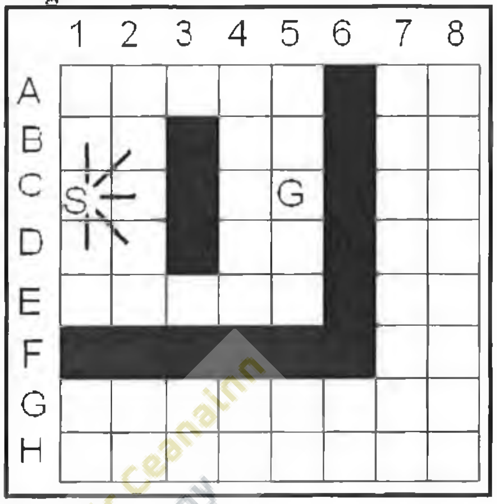
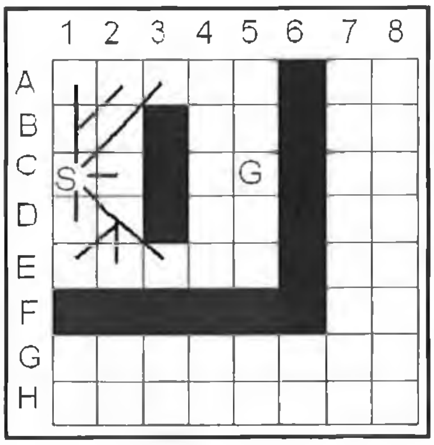


Fig 2.22 After First search of neighbours



Example: The start node is (C, 1) and the goal is (C, 5). To start off with a search begins on all neighbours of the start node, namely (B, 1) (B, 2) (C, 2) (D, 2) and (D, 1). Once the node is searched then they are placed upon a queue and the nodes are marked as being searched so they are not re-searched (to prevent back tracking). The graph would now look something like **Fig 2.22**. Node (B, 1) is taken from the queue and nodes (A, 1) and (A, 2) are searched and placed onto the queue. Then Node (B, 2) is taken from the queue and it doesn't need to search nodes (A, 1) and (A, 2) because they have been already searched. But it can search node (A, 3), this is the placed onto the queue and searched list. Node (C, 2) cannot search anything new, so that is just left. Node (D, 2) can search nodes (E, 3) (E, 2) (E, 1) (in that order) and places them onto the queue. Finally Node (D, 1) cannot search anything new so that is just left. At this point the graph will look like **Fig 2.23**.

Fig 2.23 Breadth first searching



The above process iterates until it eventually reaches the goal. If it wasn't possible to reach the goal the algorithm would end up searching the entire game world before realising that it cannot find the goal. But in a well designed game this should never happen.

From the simple example above it is obvious to see how quickly the queue can grow over time with this particular algorithm. From just one node originally, the queue grew to six nodes after just searching the original start node. If this was a search in a much larger gaming environment, the amount of resources that would be required by the algorithm would increase at each step.

A variation of the BFS (Breadth First Search) is called Bi Directional breadth first search, this is the same process shown above, but the same search starts at the goal node as well as the start node, meaning that there are 2 searches going on at the one time. Its aim is to try and cut down on the resources required by one large search, by doing two smaller searches. This search would end when a node from the start node search reaches a node from the end node search or vice versa. The advantage of this is that hopefully neither of the search areas becomes too large before a path is found.

- **Speed:** This entirely depends upon the domain that is being searched and the version of the algorithm that is being used. In general the search can be effective at finding the path if the search isn't too large. But as is the case with most games, the search area is often huge. The Bi Directional BFS doesn't prevent this either. It can only really be useful if the search is relatively straightforward and doesn't take too many twists. If the search is complex the result could lead to two very large searches instead of the intended smaller searches. Since the search is uninformed it spends as much time searching paths that will definitely not lead to the goal as it does searching paths that do lead to the goal. This is a trait of the uninformed algorithms.

- **Accuracy:** If there is a path to the goal node then this algorithm will find it. If the search was large, then it probably will not be able to find the most efficient path.
- **Usage:** Not really used in modern games due to better quality algorithms.

2.19 Depth first search

Where breadth first searches all areas with equal importance, this algorithm concentrates all efforts on one area until it either finds the goal or it finds out it cannot find the goal in that area so then it concentrates upon a different area. A useful analogy is a tree that is looking for a water source with its roots. It will send out one root to an area to see if there is water and if it doesn't find it, it will retract the root and start again but it will go to a different area. This is how depth first search works, if it was a tree search that we where doing it is the equivalent of searching on branch of that tree to a predefined depth (depth is the level of the tree that it can go to) and if it finds nothing, then it can search another branch of the tree to that depth again. The depth of the search is set to a predefined level so that the search doesn't take more time than it should. But this predefined depth doesn't always go far enough into the search tree, which means that the goal may never be found. Or it may be too long for the search, which means that there may be resources wasted in finding the goal node on other branches within the tree.

The algorithm works by starting at the start node, it then searches all the descendants of that node to a cut off (the depth of the search). As each descendant is searched it as added to a memory stack, but this queue is not the same type of queue that was used with the breadth first search. The search is prevented by back tracking on itself by giving each node a length value from the start node, if the length is shorter than the current node then it is back tracking and the search doesn't take that node into consideration. It continues this until it either finds the goal node in that branch or it reaches the start node again and then it searches another branch in that tree. If it has no more branches to search then the depth of the search is not deep enough to reach the goal so the search fails.

Fig 2.24 Depth first searching

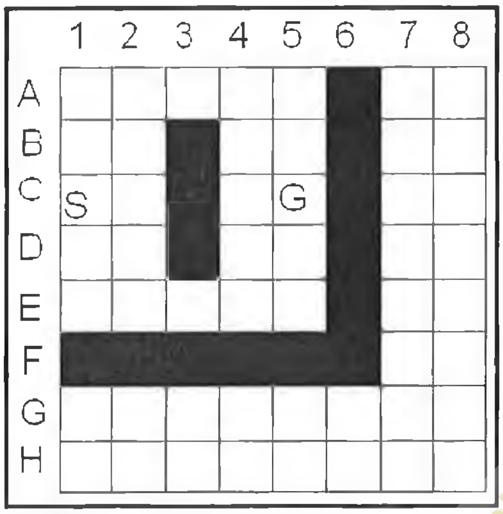
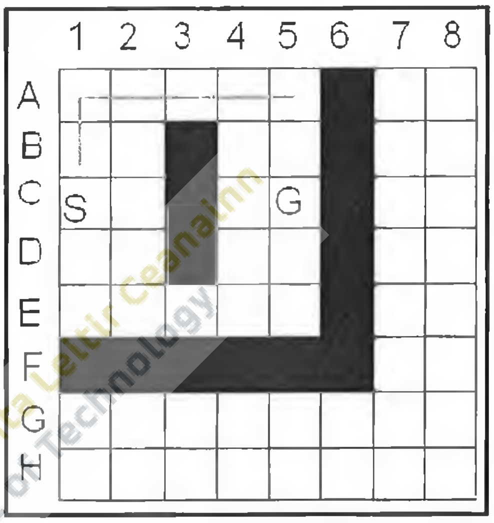


Fig 2.25 Depth first searching



For the example the depth of the search will be set to 6 (the shortest distance between the start and goal nodes) diagonal moves count for 2 depths with this type of search except for any diagonals from the start node.

It starts off by searching Node (B, 1), this node is placed on the stack and is given the length 1 (distance from the start node). It then begins to search any nodes that are connected to namely Nodes (A, 1) (A, 2) (B, 2) (C, 2) and (C, 1). But it only searches node (A, 1) at the moment, the node is placed on the stack and assigned a length 2. From node (A, 1) we again search the nodes surrounding it and the first node to come is node (A, 2). This again isn't the goal node so it is placed on the stack and assigned length 3. The search continues again from node (A, 2) and the first node to come is node (A, 3), this is assigned length 4 and placed on the. Likewise with node (A, 4) (length 5), but at node (A, 5) it has reached the depth of its search (depth is 6) so it cannot search any further along this particular path. At this point the graph would look something like **Fig 2.25**. It has to "pop" the last node (A, 5) off the stack and search the previous node namely node (A, 4). At this node it has already searched node (A, 5) and it cannot search node (B, 5) because the diagonal takes 2 searches (making the depth 7 which cannot be allowed). It searches node (B, 4), it doesn't find the goal node and it has reached its depth so it has to pop node (B, 4) from the stack and search node (A, 4) again. There are no more node to search at (A, 4) (it can't search node (A, 3) because it has a length shorter than its own length), so it has to pop the node from the stack and go back to node (A, 3). From node (A, 3) we can search node (B, 4), this is placed on the stack and assigned length 6. At node (B, 4) we cannot search anymore because of the depth, so node (B, 4) is popped and we go back to node (A, 3). There is one more node to search (B, 2) here (other than nodes that have a smaller length, so we cannot search them); again this is beyond the depth so we would just end up back at this node. There are no more nodes to search here so (A, 3) is popped and we go back to node (A, 2). From here we haven't searched node (B, 2) yet, this is placed into the stack and assigned length 4. From (B, 2), the first node to appear is node (A, 3). This will push the depth to 6 so we would just end up back at node (B, 2). Again from here we haven't searched node (C, 2) so this is pushed onto the stack and given length 5. From (C, 2) we can search nodes (D,1) and (D, 2), but since (D, 1) is a diagonal move which would push the depth to 7 we cannot do this. Node (D, 2) is searched and placed onto the stack and assigned length 6. At this point the graph would look like **Fig 2.26**.

Fig 2.26 Depth first searching

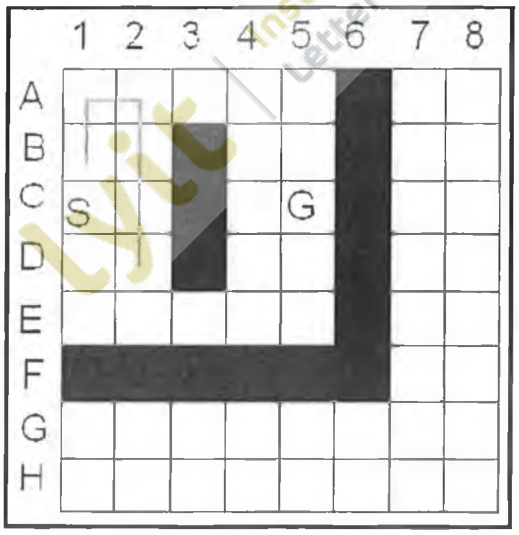
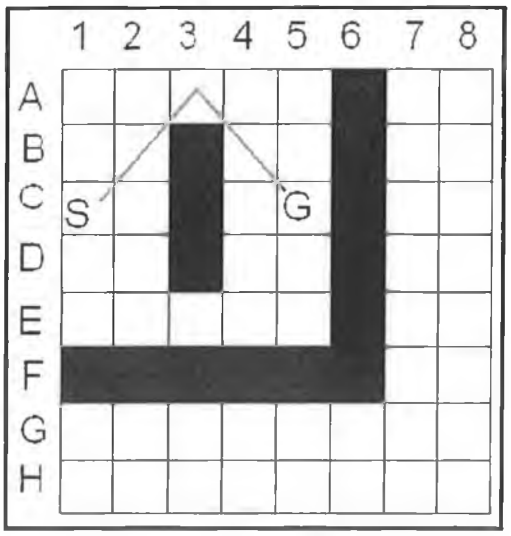


Fig 2.27 Depth first searching



From here we cannot search anymore valid nodes so (D, 2) is popped. Back at node (C, 2) again there are no nodes, there is a similar situation at nodes (B, 2), (A, 2) and (A,1). So now we are back at node (B, 1) again and the search continues from (B, 1) to node (A, 2) which is placed onto the stack with length 3 (because of the diagonal). Again from (A, 2) we can go to any of the nodes mentioned above for (A, 2) since it has the same length it is known that none of them will work so therefore wont be documented, but the algorithm will still search them, and would eventually end back up at node (B, 1) and from here it goes onto node (B, 2) and goes on another wild goose chase looking for the goal. Eventually it will go back to the start node again and another branch is taken. And the next in line is from the start node to node (B, 2) which is given a length 1 because it is from the start node. Through similar searching this branch will give the path and it is shown in **Fig 2.27**.

There is a variation of depth first search called iterative depth first search (IDDF), this is the same idea but its depth at the start is 1, allowing it to search one level down from the start. If the goal node isn't found at that level then it simply increments the depth to 2, allowing it search the level down from the previous level. This process continues until the goal node is found. It may increase the amount of time searching at the beginning if the goal is further away, but if the goal is close to the start then it will find it quicker. Also it will find the goal node is there is a path to it, it might just take some time.

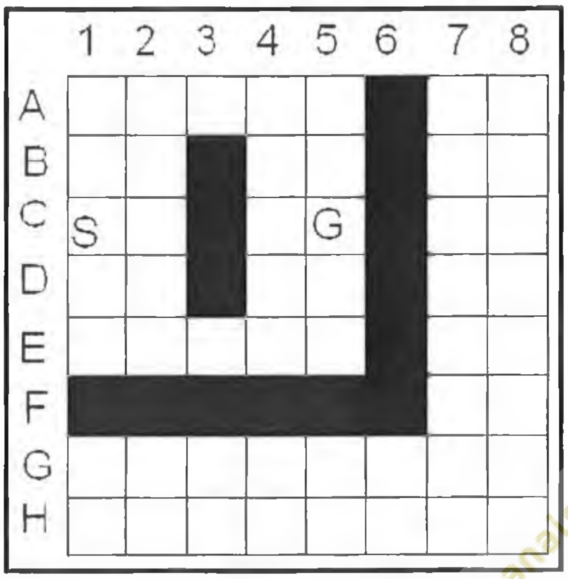
- **Speed:** Like breadth first search, depth first search's speed depends upon the complexity of the search to be completed. Again it is useful for searches where the goal node is a short distance away since it doesn't have to go very far to find it. IDDF is especially useful for this. Like breadth first search it falls over when completing searches over long distances, even more so than breadth first search. The algorithm would waste resources and time searching routes to their entirety or the depth of the search that will definitely not lead it to the goal. It gives the same priority to routes that aren't on the path to the goal as routes that are. It would be a wiser algorithm if it could select paths which would lead it to its goal, but then that would make it an informed algorithm. There is a version of DFS that does try to emulate this with varying degrees of success.

- **Accuracy:** If there is a path to the goal, then the algorithm will find it. And again like Breadth First Search, it may not find the most efficient route, especially for longer searches.

2.20 Dijkstra's algorithm

This informed algorithm is quite similar in methodology to breadth first search. It uses the same flood the area technique to encompass obstacles and eventually find the goal node. But the methodology behind it is more efficient and will find the shortest path available and it can take into account terrain cost e.g. rough terrain in a game is harder to pass than easy terrain.
The algorithm works by flooding the entire environment similar to the BFS but there are differences. In addition to the breadth first search functionality, what it is trying to do is to find the shortest path to every node within the search. So therefore when it does find the goal node it will be guaranteed to have the shortest path. So we need to keep a record of the length from each node that we search, to the start node. If at any point we find a shorter path to a particular node that we have already searched, then we need to update the distance of that node to the start node so that it can show that it has a shorter path to it than the one that it previously had. We also need to keep a record of nodes that we haven't visited yet. It uses two data structures, a priority queue and a list. For the priority queue that we will use, nodes will be ordered by the lowest length first and items will be removed from the front of the queue, this is sometimes referred to as the open list or queue. We also need a list; this is a random access data structure that allows elements to be placed and removed anywhere within it. We will use a list data type for this, it will contain nodes that have been totally processed, and will be used to make sure that back tracking doesn't occur and if there is a shorter path discovered to a member of the list then it can be removed to the priority queue so it can update any nodes that may be attached to it. This is often referred to as the "closed" list.
To explain it better again we will use the same example as for the other algorithms.

Fig 2.28 Dijkstra algorithm



Using the same grid again for the example of the Dijkstra algorithm, we start off at node (C, 1), this node carries out a search of all it neighbours and adds them to our priority queue. Nodes (B, 1), (C, 2) and (D, 1) are assigned a length of 1, while nodes (B, 2) and (D, 2) are assigned length 2 because of their diagonal move. Within the queue the items are sorted by their length meaning that the nodes (B, 2) and (D, 2) are at the back of the queue, but since node (B, 2) was placed in first it will be before node (D, 2). The start node is now processed and placed on the processed list and cannot be searched again. The first item off the queue is node (D, 1), the neighbours of this node is now searched. (E, 2) is placed on the queue with length 3 (diagonal move) while (E, 1) is placed onto the queue with length 2. The node (D, 1) is now processed and goes on the processed list; the search goes on to the next node.

Fig 2.29 Priority queue

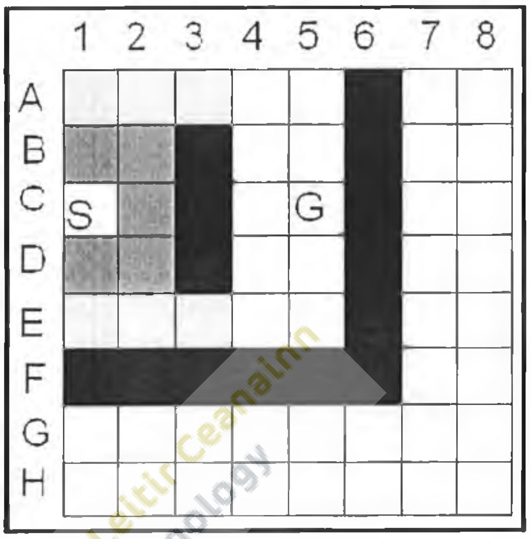
Priority Queue
C, 2 length 1
B, 1 length 1
B, 2 length 2
D, 2 length 2
E, 1 length 2
E, 2 length 3

The queue at this point should look like **Fig 2.29** so the next item off should be node (C, 2). This tries to search its neighbours but it can't do anything with them because they are a shorter distance away from the start node than the length 2 that would have been assigned to them. So nothing is done at this stage and the node (C, 2) is placed onto the processed list. Next node off is (B, 1) from here we search nodes (A, 1) (length 2) and (A, 2) length 3 and they are placed onto the priority queue. The next node off is (B, 2) and we search its neighbour (A, 3) first, it is given a length 4 and placed onto the priority queue. Next off the priority queue is node (D, 2), it tries to search (E, 2) but cannot because the path is not shorter than the existing one. It then proceeds to (E, 3), which is placed on the priority queue with the length 4. This node is then placed onto the processed list. The next search is on node (E, 2) but there is no change since it already has the length of 3 assigned to it so this path is not shorter than the one that exists.

Fig 2.30 Priority queue

Priority Queue
E, 1 length 2
A, 1 length 2
E, 2 length 3
A, 2 length 3
A, 3 length 4
E, 3 length 4

Fig 2.31 Dijkstra's algorithm

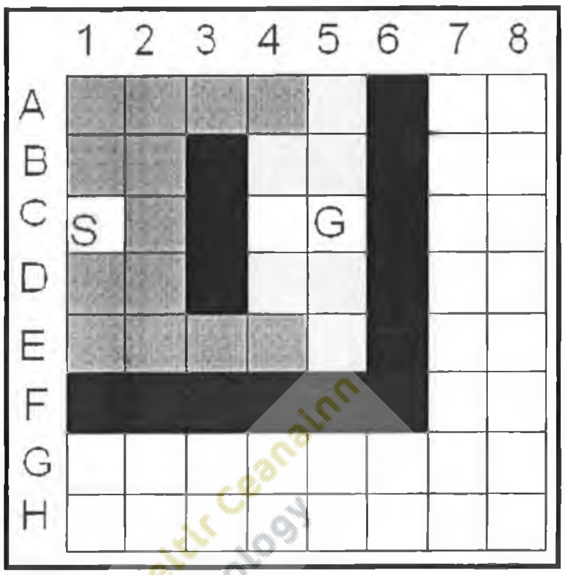


At this point the priority queue looks like **Fig 2.30** and the graph looks like **Fig 2.31** where the dark grey nodes are processed one and the light grey ones are unprocessed ones that are on the priority queue. From here the next node out is (E, 1) but nothing can be done because of the length of the neighbours that surround it. So it is simply placed onto the processed list. It's the same with the next 3 node to come out (A, 1), (E, 2) and (A, 2) which is just placed straight onto the processed list. The next node is (A, 3), which searches (A, 4) and (B, 4) and places them on our priority queue with lengths 5 and 6 respectively. (A, 3) is then placed onto the processed list. Next out is (E, 3), this places (D, 4) (length 6) and (E, 4) (length 5) ((E, 4) is placed before (A, 4) because it has the same length as the first node in the queue) onto the priority queue, while (A, 3) is placed onto the processed list. And the priority queue should look like **Fig 2.32**.

Fig 2.32 Priority queue

Priority Queue
E, 4 length 5
A, 4 length 5
B, 4 length 6
D, 4 length 6

Fig 2.33 Dijkstra's algorithm



Now (E, 4) is popped off the queue, it searches and places onto the queue nodes (E, 5) (length 6) and (D, 5) (length 7). Next is node (A, 4) which places nodes (A, 5) (length 6) and (B, 5) (length 7) onto the priority queue. (E, 4) and (A, 4) are now processed and the graph should look like **Fig 2.33**. Next node off is (B, 4) from here we can search nodes (C, 5) (length 8) and (C, 4) (length 7). Node (C, 5) is our goal node, but before we can determine that this path is the best we need to check other paths until the node is at the front or our priority queue. (D, 4) is next but it can't beat the length of any of its neighbouring nodes so it is placed onto the processed list and the priority queue should look like **Fig 2.34**.

Fig 2.34 Priority queue

Priority Queue
E, 5 length 6
A, 5 length 6
D, 5 length 7
B, 5 length 7
C, 4 length 7
C, 5 length 8

Finally the search covers each of the last nodes, all of which cannot find a shorter path to any of the nodes that are unprocessed and are its neighbours. So eventually we are left with the goal node (C, 5). So now we have the shortest possible path from the start node to the goal node.
There is also a bi-directional variation of the Dijkstra algorithm which works in the same way as the breadth first search method. And again it falls over with the same problems as the breadth first search.

- **Speed:** Since the algorithm has some of the functionality of the breadth first search algorithm it inherits its lack of understanding about the game environment. Like its predecessor, it will place equal priority over routes that will not lead to the goal and routes that will mean time and resources are wasted searching routes that will never get anywhere and hence it is known as a greedy algorithm. Greedy, because it works by trying to find the shortest path to a point, if it did find a shorter path to certain point, it would have to recalculate any node that may have been joined to that point, again creating more recalculation.
- **Accuracy:** This algorithm is guaranteed to find the optimal solution to the search proble, and as such is highly accurate.

2.21 Best first search (BFS)

This algorithm search technique is very similar to the breadth first search, in that it explores all nodes surrounding the current node. But unlike breadth first search which places priority on all routes, this algorithm prioritises routes that lead to the goal node.

Since the algorithm prioritises the route to the goal node, it achieves this using another algorithm within the BFS algorithm to calculate a heuristic. The heuristic is the estimated cost or distance of moving from the current node being searched to the destination node. The heuristic is examined in detail in section 2.23 of this document. If the graph was populated with obstacles, it would the effect calculation of the heuristic as itignores them. The idea being the smaller the heuristic, the shorter the search is away from the goal node so the search will progress generally in that direction. The main problem with this technique is that it doesn't pay any attention to obstacles or terrain cost. So with this algorithm, it will not always find the most efficient path to the goal. But at least it will find a path and another benefit is that it is relatively simple compared to other algorithms so this means that the algorithm itself takes up less memory and processor time.

Like the Dijkstra algorithm, it has to maintain two queues, one for nodes that it has checked (closed), and one for nodes that it is going to check (open). The open queue is also a priority queue like the one used with the Dijkstra, except this one orders its elements by the heuristic (estimated length to the goal node). With the element with the smallest heuristic being the first to be popped off. At each node a search is carried out on all the nodes around it. They are all placed onto the priority queue and because of the queue, the shortest heuristic will come out first. This is then the next node to repeat the search process on again. This continues over and over until the algorithm finds its way to the goal node.
We consider the same example problem used to examine other algorithms; however Instead of calculating the heuristic by using a mathematical formula as detailed in section 3.2 of this dissertation, the heuristic will be set to the actual distance from the current node to the goal node. Horizontal and vertical moves will count as a 1 and a diagonal move will count as a 2 for this example. This is to both simplify the example and show how the algorithm works rather than the heuristic.

Fig 2.35 Best first search

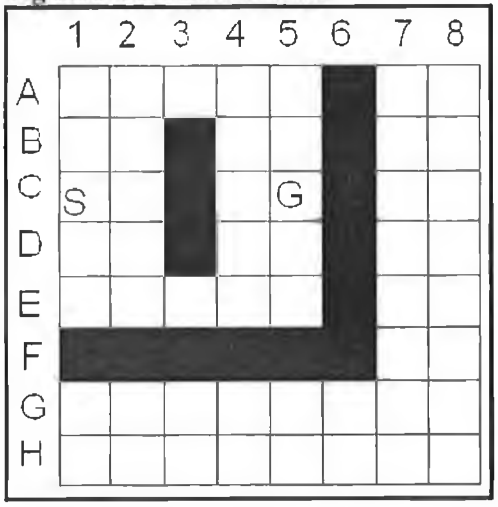


Fig 2.36 Priority queue

Priority Queue
C, 2 Heuristic 3
B, 2 Heuristic 4
D, 2 Heuristic 4
B, 1 Heuristic 5
D, 1 Heuristic 5

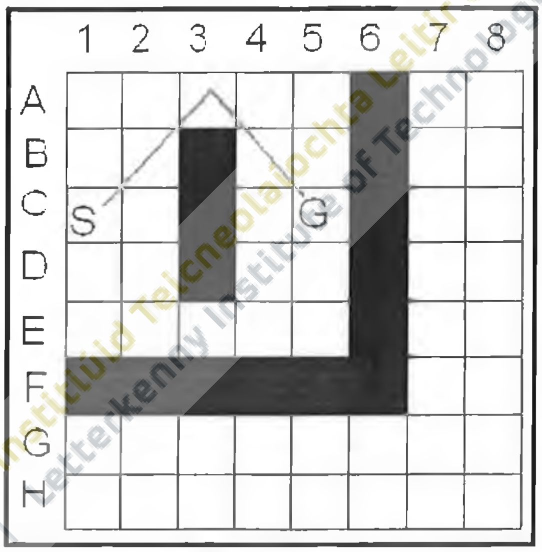
Fig 2.37 Priority queue

Priority Queue
A,3 Heuristic 4
D, 2 Heuristic 4
A,2 Heuristic 5
B, 1 Heuristic 5
D, 1 Heuristic 5
A, 1 Heuristic 6

Again starting from node (C, 1), all its neighbours are searched and placed onto the priority queue. Nodes (B, 1) and (D, 1) have a heuristic of 5 (remember that it doesn't take obstacles into account). Nodes (B, 2) and (D, 2) have a heuristic of 4, and Node (C, 2) has a heuristic of 3. (C, 1) is placed onto the closed queue and the open (priority queue should look like **Fig 2.36**. Node (C, 2) is removed and searched, it cannot move through the obstacle and the all the nodes neighbouring it are of a larger heuristic so it cannot search them, so it is placed onto the closed queue and the next shortest node (B, 2) is removed from the open queue to be searched.

It searches (A, 1) (heuristic 6), (A, 2) (heuristic 5) and (A, 3) (heuristic 4) onto the priority queue. (A, 3) is at the front of the queue now since it ahs the same heuristic as the current front (D, 2) it is simply "pushed" onto the front of the queue. And the queue should look like **Fig 2.37**. The next node out is (A, 3), this searches (A, 4) (heuristic 3) and (B, 4) (heuristic 2), these are placed onto the open queue and the node (A, 3) is placed onto the closed queue. Next off is (B, 4), this searches (A, 5), (B, 5) (C, 5) (our goal node) and (C, 4), once it has finished its search and the node (B, 4) is placed onto the closed queue, the search is finished as the algorithm has found the shortest path to the goal, which looks like **Fig 2.38.**

Fig 2.38 Best first search



- **Speed:** Aside from the calculation of the heuristic, there isn't too much computational strain generated by the algorithm.
- **Accuracy:** The algorithm is guaranteed to find a path to the goal node, however it is not guaranteed to be the most efficient path. It puts priority on searching in the direction of the goal node unaware of what is actually going on around it. The algorithm could create a path into a total dead end as long as it is heading towards the goal node. It would then be forced to backtrack upon itself to try and find another way.

2.22 A* (A-STAR) Pathfinding:

"The best established algorithm for the general search of optimal paths is A"*

(Stout, 1997)

The most common pathfinding algorithm used in contemporary computer game development is A*. Therefore it was felt that this particular algorithm warranted a lengthy review.

The aim of this and the following sections in this chapter is to investigate if the claims about A* can be substantiated in research with relation to computer games. There will be a discussion of the low level operation of the algorithm and a comparison between A* and other algorithms discussed in the previous chapter

A* is an example of an informed algorithm (section 2.16).

A* is an evolution of a number of previous pathfinding algorithms, most notably the Dijkstra algorithm (section 2.20) and the Best First Search (section 2.21) algorithm. It uses the tracking of the previous path (or the path taken to the current point within the search algorithm) taken from Dijkstra. It combines this with the heuristic estimate of the remaining path (estimated distance from the current position to the goal) as seen in the Best First Search method.

Because of the algorithm's use of a heuristic it creates an estimate of the distance from its current position to the goal, allowing it to make more intelligent decisions about where it will move next.

2.23 The heuristic

The heuristic is a number which allows the pathfinding algorithm that uses it to make more intelligent decisions about where to move next. As mentioned before, the heuristic is used by an algorithm to estimate how far away it is from its goal node. This totally disregards any environment that the agent carrying out the pathfinding might have to traverse; it is the shortest possible route between the current location and the goal even through any obstacles that may be in the way.

The heuristic is not the only method that A* uses in order to make its decisions as will be shown later. But if A* didn't make use of the heuristic, it would be the Dijkstra algorithm. This shows how A* is an evolution from Dijkstra and previous algorithms, since A* is essentially a guided Dijkstra.

The heuristic has to be carefully monitored during the A* search as it can have both positive and negative effects on the search being undertaken

"Given perfect information, A will behave perfectly"*

(Patel, 2005).

In the above quote Patel tells us that the A* algorithm can and will work perfectly as long as the information that it receives (heuristic) is perfect, since the heuristic is an estimation of the distance to the goal node, and not the actual real distance, this will lead A* to work in a less than optimal manner unless the problem is a straight line problem where the actual distance to the goal would be the same as the heuristic. For this reason, the heuristic needs to be monitored as closely as possible since if it was extremely high, then A* would deteriorate to a breadth first search. We need to be able to find the best heuristic available in order for A* to be able to work in the most 'near perfect' manner as possible. An important trade off is to try and accomplish a perfect balance between speed and accuracy. We can increase the performance of the algorithm as long as accuracy isn't an issue. This could actually be a desired quality in pathfinding as it would allow for some variation in the game play. It is possible to use precompiled exact heuristics within a game, this totally removes any computations that are associated in finding the heuristic, and this is not feasible for most game maps due to the size and complexity of most modern game maps.

Unfortunately due to the complexity of maps and search spaces in computer games, the heuristic generated will not in general be the same as the actual distance. This affects the performance of A*

There are three heuristic methods based on the following metrics.

- **Manhattan distance:** This is classed as an inadmissible method, meaning that it is likely to over estimate and fail to find the shortest path, but it is fast. Its name stems from Manhattan in New York where the buildings are built up into blocks and movement between these blocks can only be done in a horizontal or vertical basis. Similarly it applies this to the gaming grid, only allowing horizontal and vertical movement within the grid.
  From its description, it is obvious that in most situations it will overestimate the distance and possibly cause the A* algorithm to fail in finding the shortest possible path. The only positives about this algorithm are that it is easy to use, understand and implement within A* and that it is quick in finding a heuristic.

- **Diagonal distance:** This method is slower than the Manhattan method, but it is more accurate and is an admissible algorithm. There are two methods within this. One which considers a diagonal movement to be the same cost as a horizontal or vertical movement. And one that doesn't and places additional cost on a diagonal movement.
  The simpler of the two methods is the one which doesn't place any cost on the diagonal movement. For accuracy the additional cost is required on the diagonal, it's less than the cost of moving to the space via horizontal or vertical movement but it is still more than a single movement from either a horizontal or vertical move. Typically the cost of the movement is around the 1.6 value.

- **Euclidean distance:** This is the most expensive computationally of the three methods, and it rewards with the greatest accuracy. It is essentially the straight line distance between two points. The formula itself has a square root operation in it, this accounts for some of its expense and sometimes it is even removed from the method in order to make the pathfinding algorithm faster. This removal however would turn algorithm into an inadmissible heuristic, making it overestimate all searches. An inadmissible heuristic would be a heuristic that would cause the pathfinding process to work incorrectly.

Even with the described methods, the heuristic can still cause problems within the A* algorithm in games. One of the most common is encountered when several search options are found to have the same heuristic values; there is no prioritisation over which option to search. This matter will be addressed later in the document.

2.24 The mechanics of A*

Since A* is a combination of the Dijkstra search algorithm and the best first search algorithm, we already know some of the main parts of the algorithm.
Like the best first search algorithm it has two lists, both named similarly to its predecessors OPEN and CLOSED. Both do exactly the same job as they did in the best first search algorithm, the OPEN list stores nodes that need to be checked out and the CLOSED list stores nodes that have been checked out. The Dijkstra algorithm had only one list that was used to store nodes to be checked out. Within A* this is the OPEN list that stores nodes that are waiting to be checked out.

Also like the best first search algorithm, the OPEN list is a priority queue, meaning that items on the list are stored via some kind of rank. Within A*, they are stored by the score of the nodes. The score is generated by the evolution of A* from the previous algorithms or the F value as it is more commonly known.

$$\mathbf{F = G + H}$$

The above equation is used to calculate the final rank of each node. F is the cost of the node; it is generated by adding G (the cost to move from the start node to the current node (taken from Dijkstra)) and H (the heuristic (taken from best first search)). This cost is then associated with the node and it is placed upon the OPEN list. As mentioned before there is a chance that there maybe two or more F values that are the same. To prevent A* from searching unnecessary paths we can choose to also rank them by the H or G value.

From the start node, A* searches all its neighbouring nodes and assigns them values based upon the formula shown above. These are then placed on the OPEN priority queue and the original node is placed upon the CLOSED queue. The search then moves to the first node on the OPEN queue which should be the one with the lowest F value. From here it searches neighbouring nodes, updating them if a shorter path has been found to that node (Dijkstra), updating the score of the node also. The searched node is then placed upon the CLOSED list and the first node off the OPEN list is got.

A* repeats this loop until the goal node is found. From the description it is easy to see A*'s origins in other algorithms. But what sets it apart from these other algorithms is examined next.

Perhaps most importantly A* doesn't effectively backtrack upon itself when it hits a dead end. Like A*, best first search will always try to head towards the goal node using the heuristic, meaning that both will blindly walk into dead ends without question. However, best first search will create the path into the dead-end and then back track the path out of the dead end, making the path much longer than it really needs to be. A* avoids this because of its functionality inherited from the Dijkstra algorithm which makes sure it is on the shortest possible path from the start node. So it should never make the path as long as best first search.

An example of this was found in the application. A test bed application was developed within which pathfinding algorithms could be assessed (chapter 4). When A* and best first search run on a sample grid, totally contrasting results are generated. From **Fig 2.39** we can see that A* has completed the problem successfully and has found the shortest possible path from the start to the goal node. And from **Fig 2.40** we can see that the best first search algorithm has solved the problem also. But it hasn't solved it to the same accuracy as the A* algorithm did. It is obvious to see where the algorithm has started to explore the dead ends in the map and then backtracked when it couldn't find a way through, taking the path through the dead end. Maybe this would be a desirable effect, if the game's designers wanted to show that an agent is exploring an environment, maybe even finding some alternative routes around to the same goal node.

In terms of performance, the best first search algorithm gives us the better in this respect. Simply moving towards the goal requires little calculation apart from the heuristic. Memory wise we have our 2 lists that need to be managed. A* involves calculating the heuristic and performing additional calculation involving the H value. A* also requires more memory because the 2 lists would need to store more information on the nodes that are being searched and have been searched.

Fig 2.39 A*

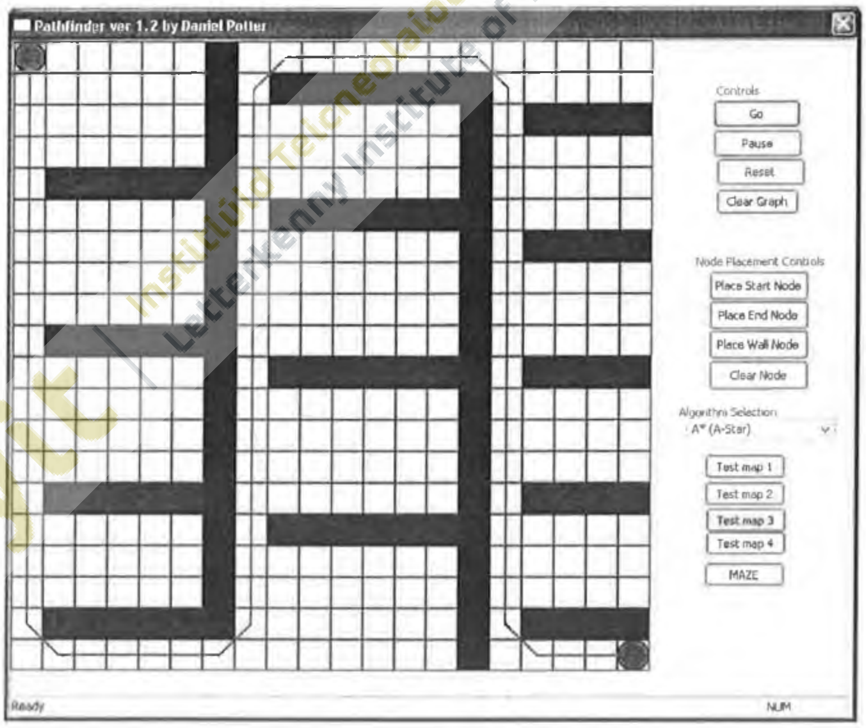
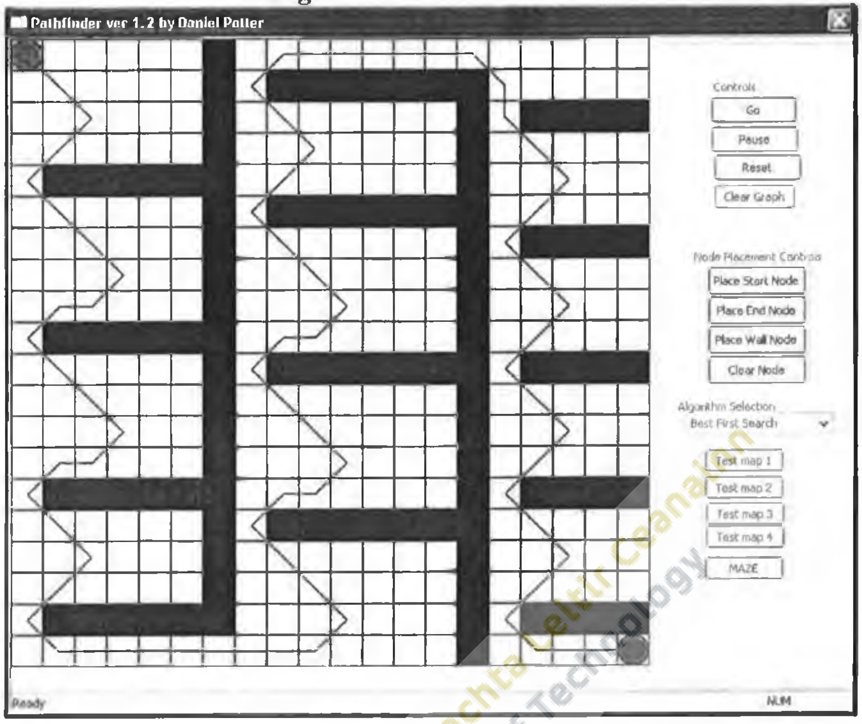


Fig 2.40 Best first search



2.25 A* limitations:

It has been determined from research that in a variety of graph situations A* outperforms other pathfinding algorithms. The next question to look at is; can we make it faster?

Now that A* has been determined to be the most accurate out of all the pathfinding techniques reviewed, what is preventing it from being the fastest? The obvious answer is processor and memory limitations of the hardware that is performing the search. The OPEN and CLOSED lists may have many thousands of entries in both of them and each entry storing data on the node. This may have a heavy impact upon memory usage.

A solution may lie in optimising the problem domain i.e. the search graph
An example of such a method is Tiered Pathfinding, which essentially is a set of multiple searches over multiple graphs. This may go against the ideal of optimising the search space by multiplying the number of graphs, but it works by having different levels of node population across these graphs, from loosely populated to densely populated. The loosely packed graph can be searched quickly to get near to the goal

node. From this point the search commences on the densely packed graph at the node where the search on the loosely packed graph finished so that the algorithm finds its way exactly to the goal node. The second search shouldn't take too long since the current position should be relatively close to the goal.

Other ways to increase the speed include the efficiency of data structures that are used to store the lists. Some are more flexible than others when tackling the demands of the algorithm. The CLOSED list only needs basic functionality since we are only adding to it. The OPEN list needs to be a priority queue; it needs additional functionality to make it work. There are several data structures that can be used for this (linked lists, sorted array), and again it is up to the designer to decide which would be the best solution.

Another method is to limit the size of the OPEN list, the idea being that when the list is full we drop the node with the worst rating, the ideal being that this node will not lead us to the best path. The danger is that there is a chance that the node dropped might be on the best possible path and thus remove the algorithm's ability to find the best path.

There is also an idea of combining the OPEN and CLOSED lists into one list and have a node flagged when it would have normally been moved onto the closed list. It shouldn't make it any faster, in fact it should slow it down since the list would have to be a priority queue of some kind, meaning that it would have to carry out all the extra sorting calculations with the nodes that are flagged CLOSED. Even though it should never have to use them since in the normal list arrangement, they would be on a totally separate list.

2.26 A-star conclusions

With reference to the original quote

"The Best established algorithm for the general search of optimal paths is A"*

(Stout, 1997)

A* is an evolution of several search algorithms but is it as powerful as the above quote suggests? From the research involved in this document the answer could be seen as being both true and false. What is the "*general search*"? As seen previously, a search within a game often depends upon what the game type. For example, the breadcrumb pathfinding in my opinion is the best way of navigating a circuit of a track in a racing game. It's inexpensive enough to allow for more AI operations to take place such as driver behaviour. But it really gives a realistic feel to the way that the car would move around the track, the algorithm would only consider the one corner at a time and not consider the entire circuit which A* would.

On the other hand, breadcrumb pathfinding would be near useless in exploring an environment where there are no paths such as a first person shooting game. A* would have a lot more success in this scenario.

So what is the "*general search*", if the search depends upon what environment it is being carried out in. It could be called a non specialised search which wouldn't require a specific algorithm such as the example above, while it could also be a search that isn't too simplistic, where the A* algorithm would be total overkill, such as a simple straight line search. It is a combination of the 2 kinds of searches mentioned; it isn't a search which requires specialisation while not being too simple.

As seen before, each search algorithm has its strengths and weaknesses which we can broadly characterise as performance versus accuracy. In general each algorithm is either one or the other. With A* it lies in the middle of performance and accuracy. When optimised to the gaming environment, it gives excellent performance since it doesn't waste resources on needless searches unlike Dijkstra, Breadth First Search and Depth First Search. While it guarantees to find the best search path as long as the heuristic is admissible, so therefore it is definitely more accurate than most pathfinding algorithms especially the uninformed algorithms. From this, we can see that A* is a general search algorithm, ideal for the "*general search*". My own conclusion on it would be

A* gives the best average performance of all search algorithms when faced with an aggregate set of searches

2.27 Dynamic pathfinding

In research to date we have considered a static graph. Within contemporary gaming context this scenario is not only possible, but likely. Agents are mobile and the graphs that they traverse are likely to change.

Such a scenario demands more of PFAs and it is in this direction that we now turn. The following section explores the possibility of dynamic pathfinding.

2.28 The dynamic search domain

The goal of dynamic pathfinding for the purposes of this document will be defined as

"The ability to alter a generated path solution in response to graph changes, and provide a more accurate path solution should it be required"

The definition states that it is "*The ability to alter a generated path solution"*, this implies an existing path has already been generated from the processing of a PFA and this path itself is dynamic. Should a change happen in the graph i.e." *response to graph changes*", then the path will have to be altered dynamically to cope with that change should that change affect a point in the generated path.

Dynamic PFAs could take a number of forms; a regular pathfinding algorithm search could be made and would generate a totally new path. This would be uneconomical in terms of processing time. It seems more sensible to modify the previous path to try and cut down on the overheads of performing a full search. If any modification of the path produces "*a more accurate path solution*" then this new path can be used by the agent that created it.

The most important aspect of the Dynamic PFAs is the idea of change within the graph. Until this point all graphs and environments considered have been static.
A graph that can accept change during run time would have to become dynamic, each node and edge would have to alter itself to accept any change that might occur in the

graph. Also since the graph is dynamic it will require a new algorithm to cope with any change to try and make a better path (if possible) using the previously generated path.

The area of Dynamic PFAs is applicable to contemporary computer games. Many titles feature destructible scenery. This idea was initially demonstrated by the THQ produced game Red Faction. Red Faction offered a system called "geomodding" to allow parts of the game environment to be totally destructible via game weaponry. This enabled the game to offer various routes through its levels and open a new tactical perspective on the first person shooter genre. Geomodding demands a Dynamic pathfinding solution.

Geomodding is effectively dynamic graph editing. The editing of the graph is achieved by either

- **Environment addition**: Adding something to the environment, that will affect the game play. E.g. Dynamic Roadblock in a driving game.

- **Environment subtraction**: Taking away something from the environment that will have an affect on the game play. E.g. Destruction of a wall between 2 rooms.

These two categories cover all events that alter the graph, at this point it is useful to define why we need to adapt to any change that could happen.

- **Environment addition**: If something was added to the environment and it wasn't added to the search graph, any agent using the graph has incomplete information about the gaming environment. Using the above example of the roadblock, an agent controlled car will not be able to recognise that the road has been blocked and will attempt to pass through the road block if it is on the best route that is available to it. It also wastes resources since the route that the agent may be trying to take has been calculated even though it may not move through it.

  This is performed by making the node or edge that corresponds to any addition to the environment non navigable in the graph. This will stop any search from using the node or edge.

- **Environment subtraction**: If something was taken away from the environment and the search graph wasn't updated the agent searching the graph has incomplete information about the environment. Using the above example of the removal of a wall, an agent searching the graph will not recognise that the wall has gone and continue to traverse around the environment as usual. This is a problem since the removal of the wall could result in a much more efficient path through the gaming environment, wasting hardware resources in the process of calculating the possibly less effective route.

  This is performed in the opposite way to environmental addition, and node or edge that corresponds to the removal of an item from the game environment becomes navigable. This allows any search that takes place, to use this node in the generation of a path.

Analysis of environmental addition and subtraction makes it clear that the graph has to become dynamic in order for the agents to effectively move around the environment. Nodes and the edges that link them must also become dynamic themselves in order to define whether or not they can be searched. The operation effectively will turn an edge on or off depending upon what happens around it.

Fig 2.41 Path open

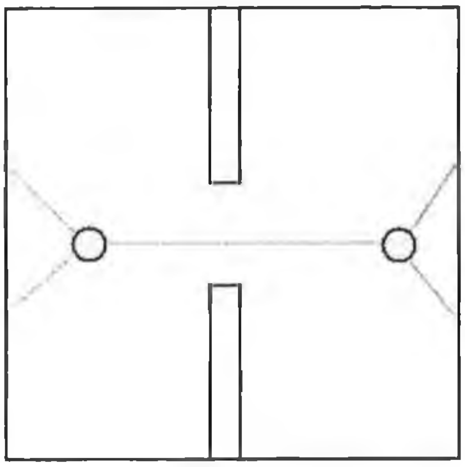


Fig 2.42 Path closed

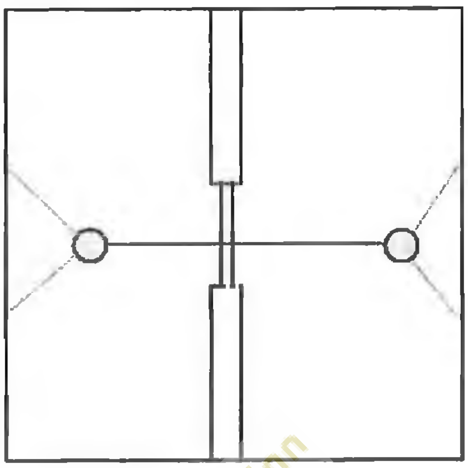


Fig 2.41 and **Fig 2.42** show instances of an environment with nodes and edges displayed; in **Fig 2.41** the gap between the obstacles is clear so free movement can take place. But should the gap be blocked (**Fig 2.42**) via some game event (e.g. environment addition) then the edge that originally joined the nodes will become dormant until the gap between the obstacles is removed (environment subtraction) and the edge can become active and be free for searching **Fig 2.41**.
The event which can block/open the gap when activated would have to dynamically change the search graph in particular the edge that traverses the gap. The event in question would probably have to be scripted in the level design; this topic goes outside of this research.

2.29 Dynamic pathfinding algorithms

Of the algorithms covered so far, none would be able to complete dynamic pathfinding since none of them can recognise and adapt to any changes that occur in the graph.

A new category of PFAs must be considered. They should at minimum;

- **Recognise the change**: Should a change happen in the graph, be it either environmental addition or subtraction, the agent executing a PFA or traversing the path that has been created previously by a PFA must be able to detect that there has been a change in the map. To make a game more realistic the agent might not detect the change unless it is within range (sight, sound) of the event which changes the graph.

- **React to the change**: Upon recognition of a change, a dynamic PFA has an immediate decision to act or not act depending on whether or not the change is affecting a point on its path. This is relatively simple with environmental addition since if the addition affects a node on the generated path then a new path needs to be calculated. The process is not applicable for the environmental subtraction as it is impossible for a computer to tell without processing whether the path is affected or not.
  Should the path be affected then a new process will have to occur in order to try and find a way around the change or possibly calculate a better path than was previously available.

There is an algorithm that satisfies the conditions outlined above, it is a dynamic version of A* called D* (dynamic A*).

2.30 The D* algorithm

D* in most instances works like Dijkstra in that it uses the distance generated to find the shortest path between two points. The difference in the algorithms is apparent when any nodes affected by the change will be researched; only modifying the selected area of the graph. Also where the Dijkstra algorithm creates a path, D* creates a signpost for each node, the continued traversal of these signposts will lead an agent to its goal node.

Like Dijkstra, D* has an open list used to store nodes that are still to be processed. So that dynamic change can be handled down the line. Nodes also have a state assigned to them, unlike Dijkstra.

Node States:

- **NEW**: This state is assigned to all nodes when the search begins.

- **OPEN:** This state is assigned to a node when it enters the open list.
- **CLOSED:** This state is assigned to a node when it has been fully processed. In the Dijkstra algorithm a node which has been fully processed would have been placed upon the closed list.

The algorithm also uses a *backpointer* (Game Programming Gems 5, 2005) to traverse from the start to goal nodes. This is very similar to the parent node functionality implemented within each node as discussed in Section 4.

The algorithm also uses the heuristic function to calculate the distance between the current position and the goal node. Unlike A* it doesn't take in the distance from the current position to the start node into account. Hence its heuristic is calculated in the same way in which the best first search method would calculate the heuristic.
There is an additional function called K(x) (Games Programming Gems, 2005) used to store the minimum of

- The heuristic before any modification is made to the search domain.
- Any heuristic value since the node x was placed upon the open list.

This would only have an affect in a search graph where the edges would have a different cost value associated with them. In the search graph associated with the application developed alongside this research, all horizontal movement results in a cost of 1 and diagonal movement with a cost of 1.4. The costs of these movements are not dynamic, so the estimated heuristic cost of moving from node X to the end node will always be the same. So the above function will be obsolete within the test bed application since they will have static values.

Using the above data a node can be classed in one of the following two states.

1. **Raise:** This state is assigned to a node when the K value is less than the heuristic generated. This is used when there is a cost increase in moving to the node being searched.

2. **Lower:** This state is assigned to a node when K value is the same as the heuristic generated. This is used when there is a cost decrease associated with the node being searched. This will happen when a shorter path has been found to the particular node.

Example from Game Programming Gems 5, on how the D* algorithm works in practice. This example does not take diagonal movement into account.

Fig 2.42 D* problem

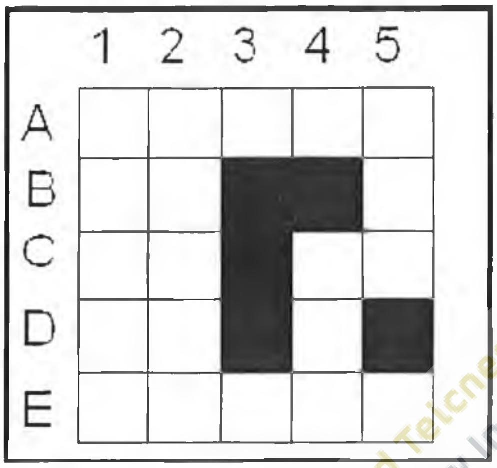


Fig 2.43 D* problem

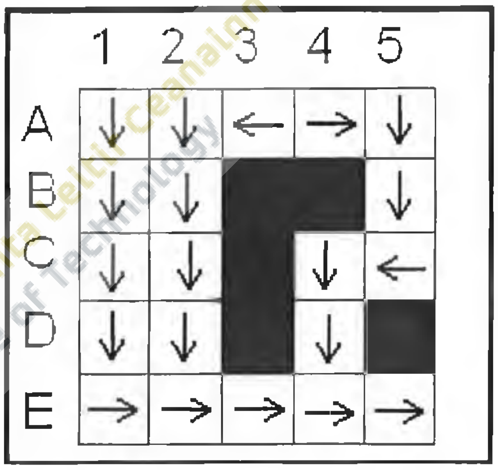


Fig 2.42 shows the initial search graph layout of nodes and obstacles. The start point is node A,1 and the end point is node E,5. Pathfinding is done via an algorithm (example doesn't specify which) and using the backpointer function the algorithm can trace the path from the start to end. **Fig 2.43** shows the layout of all the backpointers for all the nodes. Using the backpointers, the path generated would go from A, 1 down to E, 1 then right to E, 5.

Following this example, the agent will begin to traverse the path created for it and say for example when the agent it at node E, 2 an event happens which places an obstacle at node E, 3. There has been a change made in the map and the D* algorithm will need to deal with this.

The node that the D* algorithm is currently reprocessing (node (E, 2)) is checked to see if there is a change in the cost from moving from this node to the goal node. In the example it would cost more due to the obstacle (E, 3) being placed in the way of the computed path. The current node is then placed upon the OPEN queue and it is marked CLOSED. From here another search (called *processState*) is done where all surrounding nodes are recomputed to change the path until the algorithm reaches a node where the distance cost doesn't change, this means that from this node where the heuristic doesn't change there is no change in the graph to reach the end node. When there is a path change the parent node or backpointer needs to be changed to show which node it came from.

This ensures that only areas affected by the graph change are recalculated.

Fig 2.44 D* problem

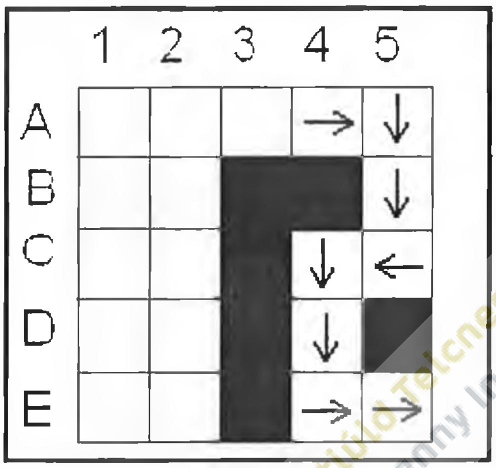


Fig 2.45 D* problem

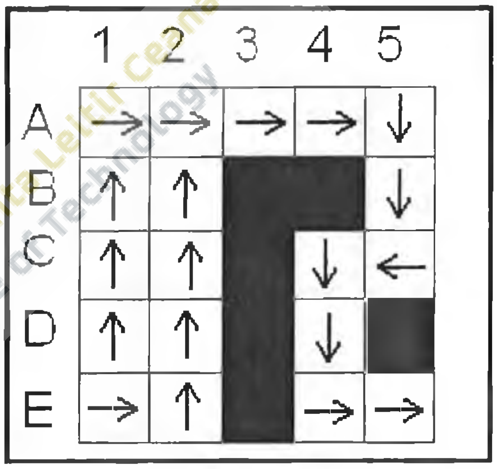


Fig 2.44 shows the nodes that need to be recalculated due to the change in the map, when the *processState* function reaches node A,4 it will see that the previous heuristic and backpointer are the same as before so therefore its search finishes here. The graph in **Fig 2.45** is produced when all the new backpointers are calculated; this graph is able to avoid the obstacle placed at node E, 3.

2.31 D* conclusion

According to the author of the example, the algorithm increases in its performance over A* with larger graphs. This would be true as D* focuses on the recalculation of affected nodes and not nodes that will not be affected by any change. A* would simply do a search from the node where the agent currently is to the goal node, with this in respect A* would be far better suited to smaller searches.

A problem that could occur with D* is that each agent will need to store their own version of the search graph since not all agents within the game will be searching from the same start point to the same end point. If every agent in the game is to store a large graph, this will inevitably increase the need for more memory for the AI in a game. Also this version of D* doesn't account for environment subtraction, this impacts on the application of this PFA to environment changing games. However it could be easily integrated into the algorithm, the environment addition above makes the heuristic rise due to longer paths. Environment subtraction would do the opposite, the heuristic would decrease.
Another important aspect of D* is the requirement to first hold an entire graph of search domains. This is fine for small graphs but, again, memory load may be heavy if every AI Agent is storing large graphs in memory.

2.32 References

- AI for game developers, David M Bourg, O'Reilly, 2004, 0596005555.
- http://www.gamasutra.com/features/19970801/pathfinding.htm, Smart moves: Intelligent Pathfinding, Accessed 19/10/05, Last Update August 1997, Bryan Stout, Article appeared in "Games developer" magazine October 1996.
- http://en.wikipedia.org/wiki/Search_algorithm, Search Algorithm, Accessed 27/10/05, Last Update 26/10/05.
- Artificial Intelligence structures and strategies for complex problem solving 4th edition, George F Luger, Addison Wesley, 2001, 0-201-64866-0.
- http://sc.tri-bit.com/Dijkstra's_Shortest_Path, Djikstra's shortest Path, Accessed 27/10/05, last update 15/12/04.
- http://www.counter-strike.net, Counter Strike Source Official Website, Accessed 3/3/06.
- AI for Game Developers, David M. Bourg & Glenn Seemann, First Edition July 2004, ISBN: 0-596-00555-5.
- The Complete Brothers' Grimm Fairy Tales, Brothers Grimm, May 1993,ISBN: 051709293X
- Rules of Play, Eric Zimmermann and Katie Salen, October 2003, ISBN 1556227353, the MIT press.
- Data structures for games programmers, Ron Penton, 2003, ISBN 1931841942, Premier press.
- http://www.microsoft.com/technet/sysinternals/SystemInformation/ClockRes.mspx, Measuring system time resolution, Mark Russinovich. Accessed 2/11/07.
- http://collective.valve-erc.com/index.php?go=hammer, Valve hammer level editor, Accessed 14/12/07.
- A* pathfinding for beginners, http://www.policyalmanac.org/games/aStarTutorial.htm, Patrick Lester, Accessed 15/11/05, Last Update 18/7/05.
- Games Programming Gems 5, Kim Pallister, "Beyond A*" by Mario Grimani and Mathew Titilbaum, 1-58450-352-1, Charles River Media, 2005.

- Games Programming Gems 5, Kim Pallister, "Advanced Pathfinding with minimal replanning cost: Dynamic A Star (D*)) by Marco Tombesi, 1-58450-352-1, Charles River Media, 2005.

3. Developing a test bed application

In section two, we looked at the mechanics of several pathfinding algorithms. In order to demonstrate the PFAs a test bed application was developed.

3.1 Application description

The research requires a program to demonstrate, via a visual metaphor, how various pathfinding algorithms solve given pathfinding problems. The graphs that the algorithms solve are user constructed. The application was also required to perform dynamic pathfinding algorithms; these changes are documented in section five and six of this dissertation.

3.2 Analysis & requirements

3.2.1 Task analysis

From the application description it was possible to derive information on several important features of the application.
The application is required to demonstrate the operation of a selected pathfinding algorithm in a given search domain/graph. It was required that the user is able to view the activities of a chosen pathfinding algorithms we requested, a visual metaphor for displaying activities to the screen. The state of the visual metaphor must correspond to the graph that is stored in memory. The graphs must also be configurable so that different pathfinding problems can be designed to test the algorithms.
From this initial analysis it was determined that the following three features will form the core of the application.

- The visual metaphor
- The editor
- The controller

3.2.2 Functional requirements

- Display the visual metaphor to the screen
    - Visual metaphor configuration should correspond to the search graph stored in computer memory.
    - Dynamic changes in metaphor configuration to take immediate affect to the corresponding graph in memory.
    - Display of algorithm activity in solving the pathfinding problem.
- Selection of pathfinding algorithm to apply to maze configuration
- The search graph is configurable
    - Through the GUI facilitate graph build by the user for the search.
- User can start, pause, reset and clear algorithm execution at any point.
- Algorithm execution is accurate to each algorithm design.
- Accurately measure an algorithm's performance in solving a pathfinding problem.

3.2.3 Performance measures

Determining the effectiveness of PFAs in given search domains is of key importance to this research. As such, several metrics for diagnosis of algorithm activity coded into the application.

When trying to define performance measures to use in the application, one can run into subjective definitions as mentioned previously in section 2.6. The following quote relates to rules of a competition for robotic mice navigating a maze. A performance measure mentioned in the quote is applicable to this research.

"*1. The time taken to travel from the start square to the destination square is called the "run" time. Travelling from the destination square back to the start square is not considered a run. The total time taken from the first activation of the Micromouse until the start of each run is also measured. This is called the "maze" or "search" time. If the*

Micromouse requires any manual assistance at any time during the contest, it is considered "touched". Scoring is based on these three parameters."

Micromouse Maze Solving competition, 10/10/06

In the context of the competition, time is seen as a key element in assessing the performance of a mouse. Therefore the application will measure the time taken for any pathfinding algorithm to find any path. This measurement will be termed 'run time'.

The Micromouse rules don't consider bad paths. A bad path would be any path carried out by the search that isn't on the final path derived when the algorithm completes. In pathfinding, creating as few bad paths as possible is vital since the processing of these bad paths takes up resources that could be used in other aspects of a game. Creating as few bad paths as possible is one measure of how efficient an algorithm is when solving a problem. As such, the efficiency performance measure was implemented in the final application. This being the number of bad paths produced by a PFA search.

Another performance measure that is critical to all pathfinding algorithms is the length of any path generated by the completion of a search. This can be measured in terms of the number of steps or nodes on the final path. This measure could be deemed as how accurate an algorithm is in solving a problem. Therefore the accuracy of an algorithm was implemented in the application as a measure of nodes on the generated path.

Much care has gone into programming each algorithm to try and make each instance directly comparable to the description of each algorithm as described in Chapter two. However, due to the unique structure of each algorithm, differences in performance could appear in the following areas.

- Unnecessary memory usage.
- Unnecessary steps or searches in each algorithm.
- Unnecessary display techniques for viewing algorithm progress.
- Unnecessary usage of programming techniques in each algorithm

The application is sequenced to avoid each these potential problems.

3.3 Application design

3.3.1 GUI layout

The following three features have been identified as important interactive features of the application and thus are implemented in the GUI of the application.

- **The visual metaphor:** Displays the graph to the user.
- **The graph editor:** Allows user to change graph.
- **The controller:** Allows user to control algorithm performance.

3.3.2 Visual metaphor design

Any graph searched by the pathfinding algorithms is made up of nodes and edges, a node being a point in the search domain (similar to a town on a map), and an edge being a connector of two nodes (similar to a road between two towns).

The manner in which the nodes and edges are displayed is unimportant as long as the meaning derived from them is unaffected; that is, to keep the graphical processing to a minimum. The chosen representation was to create a grid based environment where the nodes are arranged in chessboard fashion. This was selected because the nodes can be stored in a two dimensional array representing the grid. In this way the need for edges is removed since to move to another node means simply traversing around the array. Also from the point of view of displaying the maze, simple squares can be used to represent a node in the array. However this topic is dealt with in more detail later in the document.

The visual metaphor will also be used for input as the maze editor. Depending upon what option is chosen in the editor, clicking on squares in the maze will have the affect of the option chosen.

3.3.3 Controller design

The controller will be used to control the execution of the algorithm, it has four functions.

- **Go:** Start or resume the algorithm.
- **Pause:** Pauses the algorithm
- **Reset:** Resets the algorithm back to its start point
- **Clear:** Resets the algorithm and clears the entire environment.

For each of the above functions there was a button created in the application to allow the user to access that control.

The controller will also provide a facility to choose the algorithm.

3.3.4 Editor design

The editor is a suite of tools that allow the user to change the graph during application run time. The features of the editor are

- Add a start node (there can only be one start node within the environment.
- Add an end node (there can only be one end node within the environment)
    - Both the start node and end node have to exist before any meaningful search can take place.
- Add obstacles – the maze walls.
- Clear a node and making it a blank node.

For each of the above functions there was a button created to allow the user to change the graph.

3.3.5 Graph & node design

In the application the search graph is a large number of interlinked nodes. Each of these nodes will be an object within the application. As such it has its own private data as described below.

- Needs to know where it is in relation to the array of nodes (x and y coordinates).
- Needs to keep track of algorithm specific data, such as heuristic and distance.
- Needs to store parent data to draw final path
- Needs to be able to tell if the node has been searched or not.
- Needs to be able store what kind of node it is (clear, start, end, wall).

Each of the private data members described above will have its own get and set functions which retrieve or change the relative data as required.

Typically a graph is a large number of nodes connected by edges, these edges connect two nodes together, and a node can have any number of edges. However edges were not used for the search graph for the application as described later in this chapter.

3.3.6 Screen design

Fig 3.1 Screen shot of final application

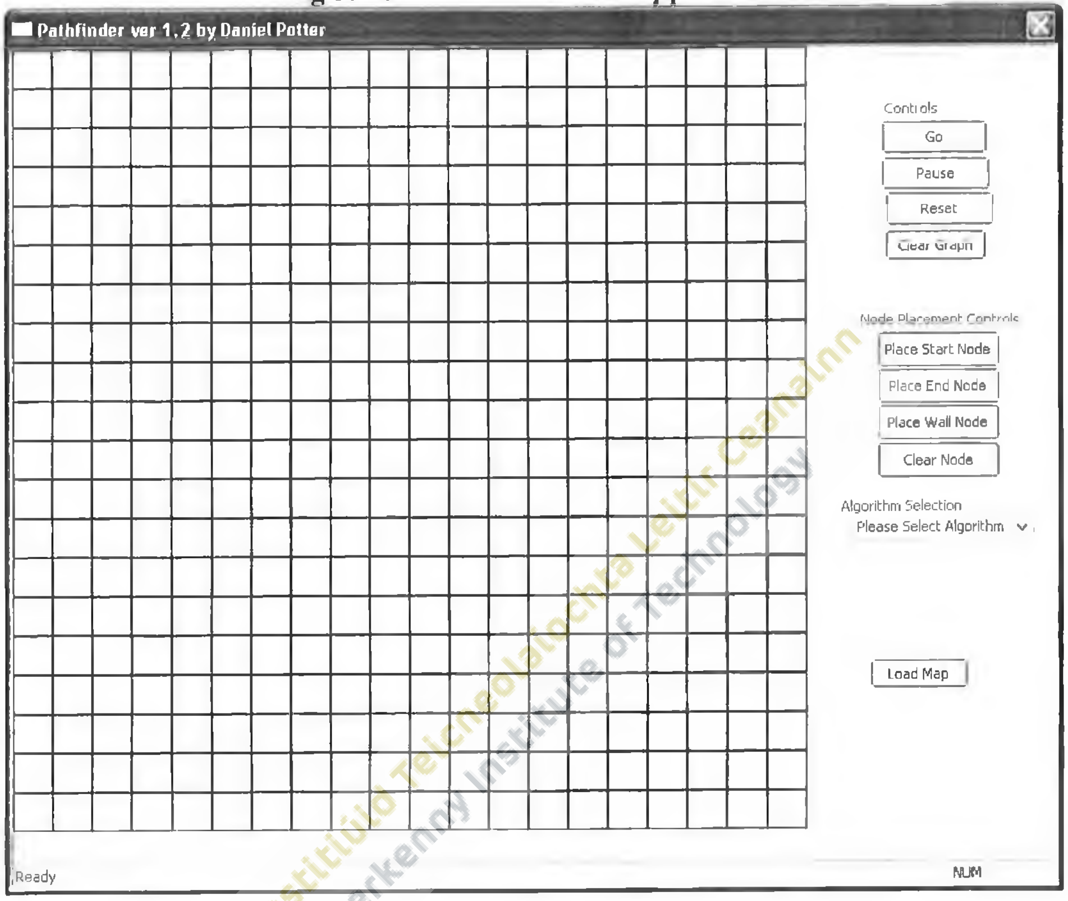


Fig 3.1 shows the final application GUI, the proportion of the window is taken by the maze which at the time of the image being taken was totally unpopulated with start and end nodes.

3.4 Implementation

3.4.1 Choice of development tool:

The choice of development environment was a major concern at this point. There are many tools that could be used to complete the task such as 2D gaming engines: Visual Basic, Java, C++.
The final decision was to use a combination of MFC (Microsoft Foundation Class) and C++ to solve the problem. MFC is a tool kit designed to make the most common actions used by a window easy to program and use. One such item is the design of forms which contain controllers (such as buttons, scroll boxes etc) and the ease of creation of event listeners which operate when a controller is activated.
Visual C++ was also the most familiar programming language to the author and would be one of the industry standards for a programming language for developing games.

3.4.2 MFC & GDI implementation

The basic framework of the application was constructed in Visual Studio; this creates a simple base application with the whole model-view-controller paradigm which MFC uses.
The wizard allows the creation of various types of MFC applications; the following was the configuration that was used.

- MFC application: the application supports MFC coding.
- Single document Application: The Application is not a windowed one =, so it doesn't allow child windows. The application needs the single window on which to display the grid.
- No Database support: The application doesn't require any databases.
- A few user interface features: Such as close buttons, thick frames.

MFC implements a model, view, controller architecture to aid the development of applications, some of the features are mentioned below.

- Model class: this is the class where the data is stored, for the application this will contain the data for the grid of nodes, and each of the implementations of the algorithms.
- View class: This class renders a UI (user interface), this can't be used to manipulate the data in the model, in the application this will be the screen design as shown in **Fig 3.1**.
- Controller class: Used to handle any events generated by the view class, this class can then change the data according to what event has occurred.

The single document application creates a very simple window with a white background. MFC allows dialog boxes to be placed within the window, these dialog boxes are created visually in Visual Studio, and they can then be placed where wanted when the window is initialised. These dialog boxes contain controls such as buttons and combo boxes that used to implement the editor and the controller.

GDI is the Graphics Device Interface, it allows the creation of simple graphics to a window. It was used in the implementation of the maze and algorithm processing since it enables the drawing of simple squares, lines and user defined shapes in different colours.

3.4.3 Pathfinding algorithms implemented

To try and provide a wide spectrum of results several algorithms were coded into the application.

- Random Bounce
- Wall Tracing
- Breadth First Search
- Dijkstra
- Best First Search
- A* (A-Star)

3.4.4 Algorithm implementation

- **Memory structures:**

All of the informed algorithms use some kind of memory structure to store data on the search that is taking place.
The main types are:

- **List:** Data storage methods which other methods mentioned below are based upon. A list is exactly what it says it is, it enables the programmer to list out items, in this case nodes in the computer's memory. A list is dynamic, meaning that it has no fixed size and enables the programmer to increase or decrease the size as required this makes it good for saving memory use.
- **Stack:** A data storage method where items are placed within a list, items are removed in Last In First Out method (L.I.F.O)
- **Queue:** A data storage technique where items placed on the queue are processed in a First In First Out method (F.I.F.O)

- **Priority Queue:** Similar to queue, but as elements are added to this type of queue they are sorted by some type of rank. So that the item with the desired rank will be the first to be processed each time.

To decrease the amount of memory used and hopefully increase the performance of the algorithm. Originally, the entire node object (including private data and functions), was to be sent whenever a function call was made requiring data from the node in question. This would create a duplicate node which takes up additional computer memory it would also need several operations to create the entire node.
With the emphasis in making the application as efficient as possible the above scenario isn't good enough. The answer was use memory pointers to each node to be sent, these pointers are created dynamically and are used as a reference to a node object. The memory pointer itself is only a reference so its memory size is small and it would only take a fraction of processing that an entire node would take.
C++ already offers coded versions of these structures so this will further simplify development.

- **Control during the algorithm performance**

The application should be able to accept input from the interface during the algorithm's execution. However, since the program is linear, the application had to be able to listen for input at each iteration of the algorithm. To further complicate the situation, forms that are used in MFC won't be able to accept input during the performance of each of the algorithms since the processing of the algorithm is the only thing that the application can do at that particular time. Input can only be accessed once the processing of the algorithm has been completed. So if a user wanted to pause the algorithm's performance, during the actual performing of the algorithm it wouldn't work.

The answer was to use threads, threads allow for multiple processing in an application. For this application, it would require the algorithm to work on a thread so that the forms can still accept input from the user.

The 'worker thread' is ideal for this purpose, as we don't want to interfere with the thread once it has started other than to pause or cancel it. It just needs to perform the algorithm and free up the MFC controls. The worker thread will simply run the algorithm in the background while the interface is free in the foreground.

- **Rewriting nodes**

Two of the algorithms offer the ability to rewrite over a node if a shorter route has been found to that node. This presented several problems, as it could entail that two versions of the node could be placed upon the queue at a given time, resulting in the processing of the node twice. This could lead to further errors when performing the later stages of the algorithm.
Something like the following pseudocode could be used to try and restore normal functionality.

```
If(newDistance < oldDistance) then
    Update Node
    Delete old node entry from data structure
    Add new node entry
endIf
```

Once this has been completed then the graphical update will need to occur where the new path is drawn onto the screen.

- **Storing the Generated path**

When any of the algorithms have finished processing a path should be generated between the start and goal nodes. In order for an AI agent to use this path it needs to be stored in memory so that any agent can follow the nodes in the path to find its goal.
This task was achieved using the list memory structure as described above.

- **Displaying the path**

  Once the algorithm has completed it would be useful to redraw the actual path that the algorithm has found, this can be used to both show the working of the algorithm and its accuracy.

  The best method to do this would be to trace back from the end of the search back to the start. Each node could have a parent variable which stores the parent which searched the node. This is not only useful for displaying any path generated by any algorithm, but also it can be used to create the list of nodes that are on the shortest path. It would work by adding each node to the front of the list as it is drawn onto the visual metaphor for the search graph.

3.4.5 Maze & graph implementation

As explained before, the maze is merely a visual representation of the search graph. And the search graph is a group of interrelated nodes and edges. If edges are to be used a node could have any number of edges or none at all.

The grid based maze implemented allows the user to edit and create an environment very simply and any environment constructed would be very similar to that of any maze. The ideal example is that of a maze from the original Pac-Man game and would be simple to replicate using the cellular automata squares.

Fig 3.2 Pac Man Maze

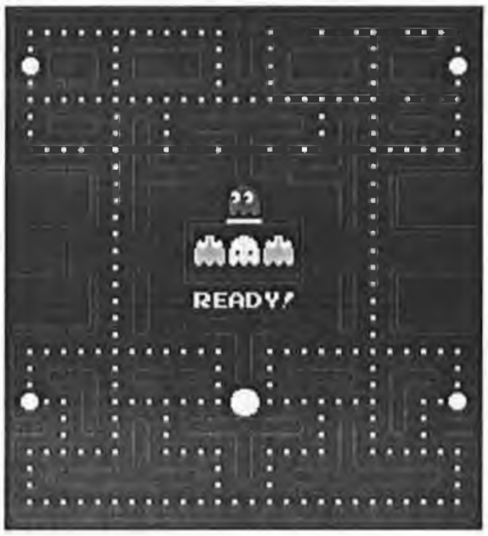
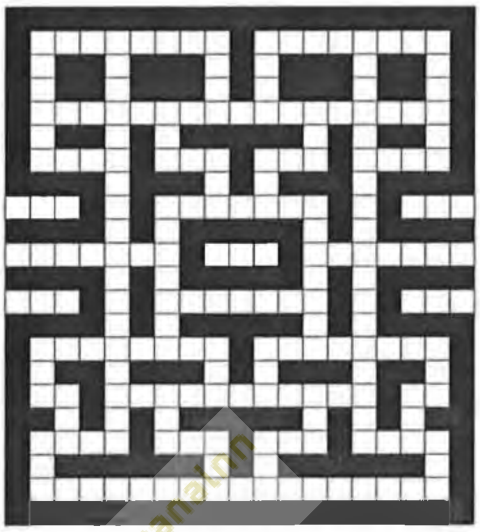


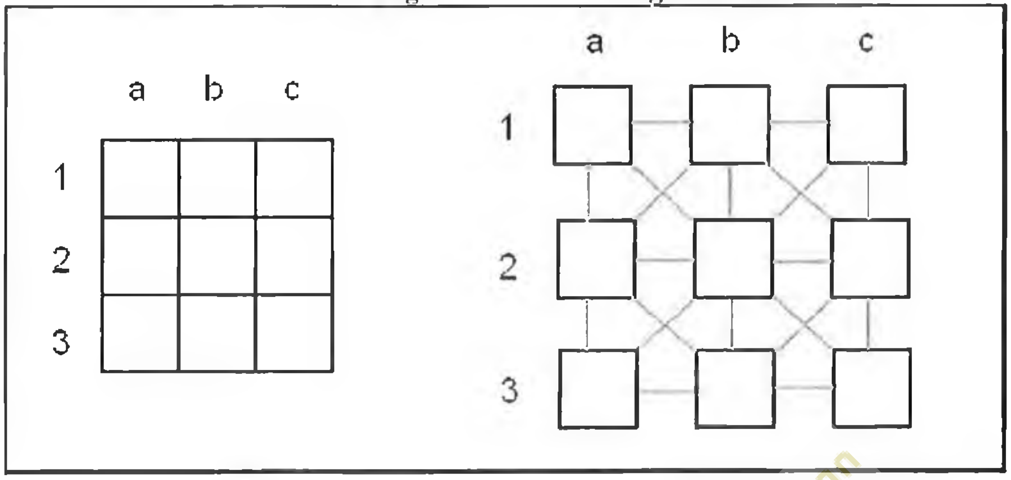
Fig 3.3 Grid based Version



Although the design for the grid based search domain implemented doesn't look like it contains edges needed to complete a search, in the theory behind it, it does.

Fig 3.3 shows grid based arrangement of 9 nodes on the left. Movement around this could be simply achieved by the increment and decrement of a pointer to a node within the graph. These increments and decrements are very similar to the way that the edges work and if the nodes where to split apart to show how the increments and decrements would affect it. It would look something like the graph on the right side of the image. The lines can be considered as edges.

Fig 3.4 Nodes and edges



A node within the environment can be in only one kind of state out of several depending upon what it contains. So far a node can only be one of 4 states which was represented by a number stored in the node class.

- Blank node: number 0
- Wall Node: number 1
- Start Node: number 2
- End Node: number 3

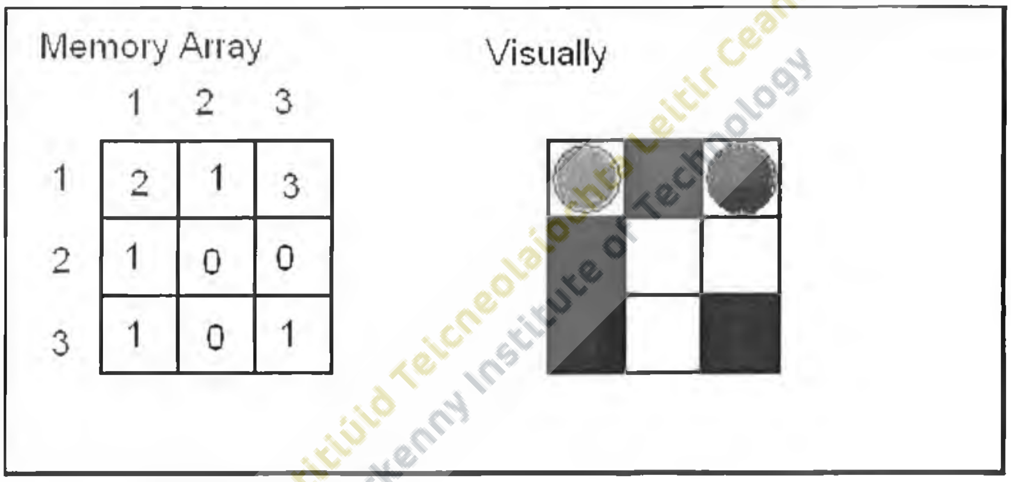
The data is stored within the array of nodes on what each contains to allow a full traversal.

In order to display the maze visually to the screen, it is necessary to use basic graphics that can draw shapes and lines so that these can be used to represent the search environment. For this the graphics device interface (GDI) toolkit was used.

Windows has a special message that is used to tell a window that it needs to redraw itself due to an update. This message is called WM_PAINT. It can be called whenever the window is moved or resized. For this application the WM_PAINT message will be used to redraw the grid. However due to the algorithms being processed in a thread it will be difficult to redraw the entire algorithm's progress at each call of the message, since data will need to be stored on every single move the algorithm has made in a search.

One of the requirements mentioned earlier was that the visual maze configuration should correspond to the search graph stored in computer memory.
As mentioned earlier the nodes will be stored in a 2-dimensional array in memory as a node object. The state of the maze within memory must correspond to the visual metaphor.

Fig 3.5 Memory array corresponding to visual maze



	1	2	3
1	2	1	3
2	1	0	0
3	1	0	1

When the application is run, a blank initialized maze is delivered, all the nodes contents in the maze are set to 0. The user can then build their own maze using the editing tools.

The visual maze and the maze stored in memory must correspond to one another for the application to work properly; making sure the maze in memory corresponds to the visual maze and vice versa is discussed later in this chapter.

When changes are made to the maze, both the visual side and the memory side must be in sync with one another. To prevent either displaying false information, the visual maze reads data from the memory whenever a change has taken place.

To redraw the maze each node in memory is checked one at a time and its corresponding node is drawn to the grid. Since memory arrays in C++ start at 0, this will be used to find the exact screen coordinate to draw the contents of the node to. For example, we want to redraw node 5, 7. Since the arrays start at 0 and not 1, the coordinate is decremented to leave 4, 6. The data is read from the maze array and then the corresponding node is drawn in the square at coordinates 200,300.

Another requirement was that during the processing of each of the algorithms, the maze should be able to display how the search is conducted in the computer's memory. This is to give the user a step by step account of how each algorithm uniquely searches the graph.

The use of the maze to display an algorithm's search was considered useful. The metaphor for the search employed was a line which was drawn from one node to another in the event of a successful search of that node from the searching node; it may not draw outside of the grid or onto node that have been denoted as a wall.
To get the right coordinates to draw the lines from one node to another, a similar method to the one mentioned above for drawing the contents of nodes was implemented. To draw a line using GDI, a source and end coordinate is required; these coordinates can be derived from the searching and searched nodes as mentioned before. To draw from the centre of the node it is a simple matter of adding 15 pixels (half the size of each square in the maze) to the x and y coordinate since the square are exactly 30 x 30 pixels in size.
This delivers an accurate visual metaphor for the activity.

3.4.5 Dynamic maze configuration

The maze editor was designed to be as simple as possible while providing enough functionality to let the user alter the maze/graph configuration. To visually display the editor tools, MFC dialog boxes containing buttons relative to the functionality it should provide.

One of the problems discovered when working with the maze editor is deriving meaningful input from the user when the maze design was taking place. To resolve this, mouse input was used to place the items on the screen, it would be a simple trap event and then depending upon what state the application is in then it would take the input and refine it. Depending upon where exactly on the screen the mouse pointer is pressed, the exact coordinates can be gained, these coordinates can then be filtered by the size of each box, thus giving us an exact reference for the exact box that had been clicked upon.

This was done by trapping the x and y coordinates of the mouse press within the application window. This would give a value totally unusable e.g. 347,609, this was converted by using a modulus operation to divide the number exactly by 30 (the length and width in pixels of each of the squares in the maze). This gave us 11, 20 for the above coordinates and would be the exact reference for the corresponding node within computer memory that the original mouse press was done on.
This refined reference can then be used to manipulate the nodes in memory and alter it depending upon the state the application is in.

The application's state will change depending on what button is pressed within the dialog box, these states effect what the application can do at a time. These states are:

- **Null state:** State where no command has been issued.
- **Place Wall State:** State where the user wants to place a wall node in the editor.
- **Place Start Node:** State where the user wants to place the start node within the editor, there can only be one start node.
- **Place End Node:** State where the user wants to place the end node within the editor, there can only be one end node.
- **Clear Node:** State where the user might want to clear any of the nodes within the editor.
- **Go:** state where the user wants the algorithm selected to perform, to be in the go state an algorithm has to be selected and both the start and end nodes have been placed.

- **Pause:** when the algorithm is performing the user can choose to pause it, algorithm will resume when go is pressed.

The state in which the program is in will change depending upon what buttons are pressed. So any state can be achieved from being in any of the states mentioned above.

3.4.6 Developing D*

For an accurate evaluation of any original pathfinding algorithm developed alongside this research it is necessary to have a comparison algorithm. For this research D* as mentioned in section 2.30, will be the comparison algorithm and therefore it was implemented in the pathfinder application.

Since the D* algorithm is dynamic it automatically creates issues for the pathfinder application which so far has only been developed for static graph algorithms.

- **The algorithm needs to be performed more than once**
  All other PFAs developed so far only require a single performance and the path is created. D* requires that one run is performed to create a map of the environment. Once a change has been made then it is this map that is altered accordingly, thus requiring a second performance for changes to be detected.

- **It doesn't create a path:**
  Again unlike other PFAs the D* algorithm doesn't create a definite shortest path for any AI agent to traverse. It creates a map of arrows where from any node on the graph if the AI agent in question follows the arrows, it will reach the end point via the shortest route. It is these arrows that are altered whenever a change takes place.

- **Requires the processing of the entire graph**
  Even when the end node is reached via the searching via D*, the processing has to continue since we will need the entire map processed in order for any accurate dynamic pathfinding to take place. Since nodes that may not have been

processed if the algorithm completed its search, may be required to calculate a new path because a change in the map may have occurred.

The description of D* in the book *Game Programming Gems 5, Section 3.8 Advanced Pathfinding with minimal replanning cost: Dynamic A Star (D*)* is entirely theory and is very often vague on how some of the mechanisms of the algorithm actually work.

For this reason the algorithm was developed from the ground up and using the description of D*. A list of processes that D* performs in order to complete was drawn up. This list was then used to develop the actual algorithm in the pathfinder program.

1. D* Searches every node on the graph even when the goal node might be found.
2. For each node it generates a "signpost".
3. Once complete, from any processed node, an AI agent should be able to follow the "signposts" to reach the goal node.
4. If a change takes place, the entire graph is re-searched; if any changes have taken place then the algorithm alters the signposts accordingly.
5. The signpost are generated by using a heuristic, the article in *Game Programming Gems* is quite unclear on what kind of heuristic is to be used.

Each one of the above requirements needed to be addressed in order for the algorithm on a whole to work effectively. The following is an analysis of each problem along with the final solution that was implemented for each problem.

1. D* Searches every node on the graph even when the goal node might be found.

As mentioned before D* ideally must search every single node in the graph to make the dynamic pathfinding as accurate as possible. The ideal solution here would be to use one of the already developed algorithms in order to perform this operation. Preferably one that can be easily adapted to perform the searching of every single node, even after the goal node has been found. At this point there is no concern over the heuristic as this is discussed at a later point.

The two possible algorithms are

- Breadth First Search
- Dijkstra's Algorithm

Both of these algorithms are discussed and explained in chapter two: pathfinding algorithms. Both of these algorithms treat every direction of a search equally, thus ensuring that the entire graph will be covered.

The algorithm that was chosen was Dijkstra's for the following reasons

- It uses the distance from the start node in order to calculate which node to search next.
- It incorporates a priority queue to select the best node.
- It has already been established that Dijkstra is similar to D*.

The final aspect is altering the termination of the algorithm so that it will stop whenever the priority queue is empty instead of when the goal node is found. The priority queue will only ever be empty when there are no more nodes to be processed therefore ensuring that every single node in the graph is done.

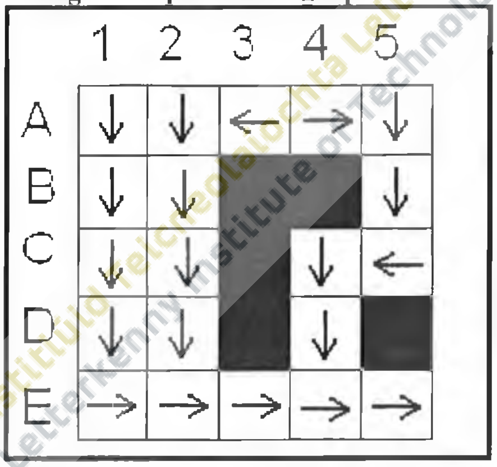
2. For each node it generates a "signpost".

The signpost is a pointer to the node attached to the searching node that is the shortest measurement from the goal node. This process will be completed in a similar way to the way that other pathfinding algorithms search the nodes connected to the searching node. It will search each connected node in turn, assigning it a measurement. When all of the nodes connected to the search node have been examined, the one which has the lowest or highest measurement (depending upon measuring method chosen) will be assigned as the node for the "signpost" to point at.

3. Once complete, from any processed node, an AI agent should be able to follow the "signposts" to reach the goal node.

Once the process outlined above has been completed for all nodes in the search domain the algorithm will have finished processing. Then each blank node in the visual maze will have an arrow associated with it. It should look like the **Fig 4.6** as shown earlier in this chapter, where the goal node was **(E, 5)** and the start node was **(A,1)** .

Fig 3.6 A processed graph in D*



From any node within the maze and not just the start node, if the arrows are followed it should reach the goal node via the shortest path. This particular algorithm only needs a goal node for processing, the start node is immaterial. This again can be seen in **Fig 4.6** as the goal node was **(E, 5)** and the start node was **(A, 1)**. The resulting graph would have been exactly the same had the starting node been any other node on the graph other than the goal node.

4. If a change takes place, the entire graph is re-searched and if any changes have taken place then the algorithm alters the signposts accordingly.

Once the algorithm has completed and we are left with a processed graph as detailed above the application is able to make changes to the graph. A change can either be the addition of a wall node (environmental addition) or the removal of a wall node (environmental subtraction).

The algorithm searches through the nodes as described above a second time and for each node it compares the measurement recorded during this search to the measurement that was created during the previous search. This function is very similar to the K(x) function as mentioned in the description of the D* algorithm taken from *Game Programming Gems 5.*

There can be three possible outcomes to the comparison of the measurements unlike the article in *Game Programming Gems,* which has two.

- **Measurement is the same as before:**
  This means that no change has taken place in the graph.
- **Measurement is less than stored measurement:**
  This means that the algorithm has found a shortened route to the node due to a change in the graph. The node is reprocessed and the change in the graph is made.
- **Measurement is greater than stored measurement:**
  This is the possible outcome that isn't mentioned in section 5.5, this will only happen when the algorithm has found a path to the node which is longer than the path from the stored node, this is due to a change in the graph.

If a change occurs in a node, this change will affect most of the nodes that it is attached to.

5. The signposts are generated by using a heuristic, the article in *Game Programming Gems* is quite unclear on what kind of heuristic is to be used.

Game Programming Gems 5 at no point specifies at what kind of heuristic is used, often simply declaring the heuristic to be *x*, with no definition of how *x* was derived.

Several methods for measuring a heuristic were mentioned in section *3.3 The Heuristic*, which are used for the A* pathfinding algorithm. As mentioned before the A* heuristic is built from two previous algorithm's measurements, those being Dijkstra's distance and best first search's (BFS) measurement.
BFS measurement is no use to this algorithm due to the fact that its measurement totally ignores terrain. If it was used it would never be able to pick up any changes that may have happened in the graph, a node will always be the same distance from the goal if obstacles are ignored.

This has an effect on the A* algorithm too since it uses the BFS measurement making that particular part of the algorithm redundant. The other part of the algorithm is the actual distance that is searched by the algorithm which is used in the Dijkstra algorithm. The actual distance of a search to a particular node can change with the addition or subtraction of obstacles from the graph. It can either be the same distance, shorter distance or a longer distance. Each of these reflects the outcomes of the K(x) algorithm described above.

In the actual development of the D* algorithm the A* method for generating the heuristic was tested, it had a negative effect on the graph generated as some nodes when processed were clearly incorrect.

3.4.7 Development issues

As the algorithm was developed and tested it became clear that there was a problem generating the correct direction for the signpost from each node. Upon commencing the search from the start node the algorithm would work as defined above. The problem occurred when it tried to generate the correct direction when the node was moving into a dead end in the graph. All nodes until the final nodes in the dead end would point towards the final node.

A graph generated using the algorithm at this point would look like the one generated in **Fig 3.7**. Although it does find the correct path to the goal node, the nodes at **(B, 3)**, **(C, 3)** and **(D, 3)** are incorrect, this could have an affect should there be a new path created and the reprocessing of nodes moves to these nodes.

The problem was that the search started at the start node, which for any of the other algorithms is essential. But as mentioned before in this section, the start node is of no use to the D* algorithm. As a result the development switched to start the searching from the goal node since it is this node that we want to search and the nature of the Dijkstra algorithm used is to find the shortest distance from the node where the searching started (in this case the end node) to every node within the graph. Upon developing this method, the algorithm performed correctly and all processed nodes "signposts" pointing in the correct direction as shown in **Fig 3.8.**

Fig 3.7 D* search problem

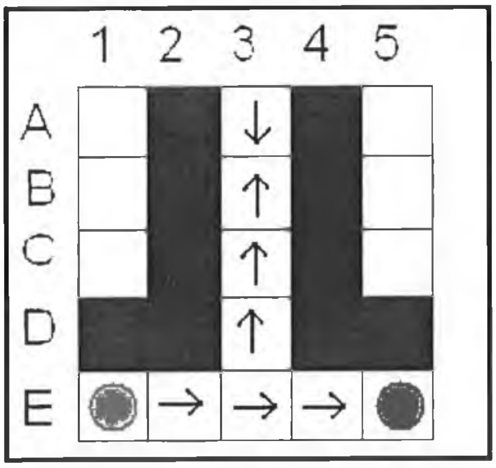
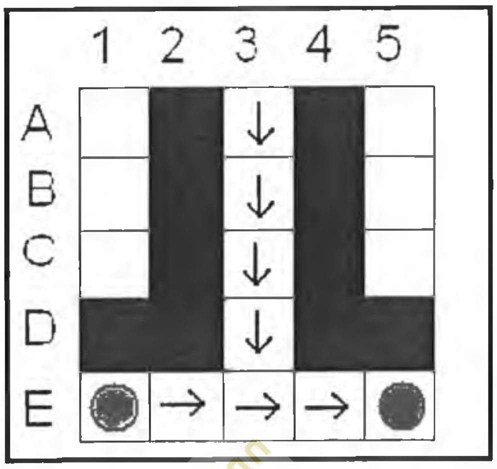


Fig 3.8 D* search problem



3.5 References

- Games Programming Gems 5, Kim Pallister, "Advanced Pathfinding with minimal replanning cost: Dynamic A Star (D*)) by Marco Tombesi, ISBN No 1-58450-352-1, Charles River Media, 2005

4. Algorithm testing & evaluation

4.1 Introduction

Having developed the testbed application to demonstrate how each unique pathfinding algorithm works, attention turned to evaluating the overall performance of each algorithm.

4.2 Algorithm Efficiency

Instead of computing Big O of the algorithms mathematically, the efficiency of the algorithms is measured using a series of tests and the results are plotted using graphs i.e. **Fig 4.14**. *Run time* is measured in milliseconds which on a computer isn't a totally accurate measurement. This is due to the system timer resolution, which for the machine on which the testing is done is 15.625 milliseconds.
The graphs show the average run time it takes to solve each numbered problem.

4.3 Test graphs

To get an accurate value for the order using big O notation, each algorithm's performance is assessed over 4 graph problems which increase in problem size. Each problem will be executed several times to get an average value for the execution. Each problem domain was created in the pathfinder application detailed in chapter four and each looks as follows.

Fig 4.1 Test graph 8 x 8 grid

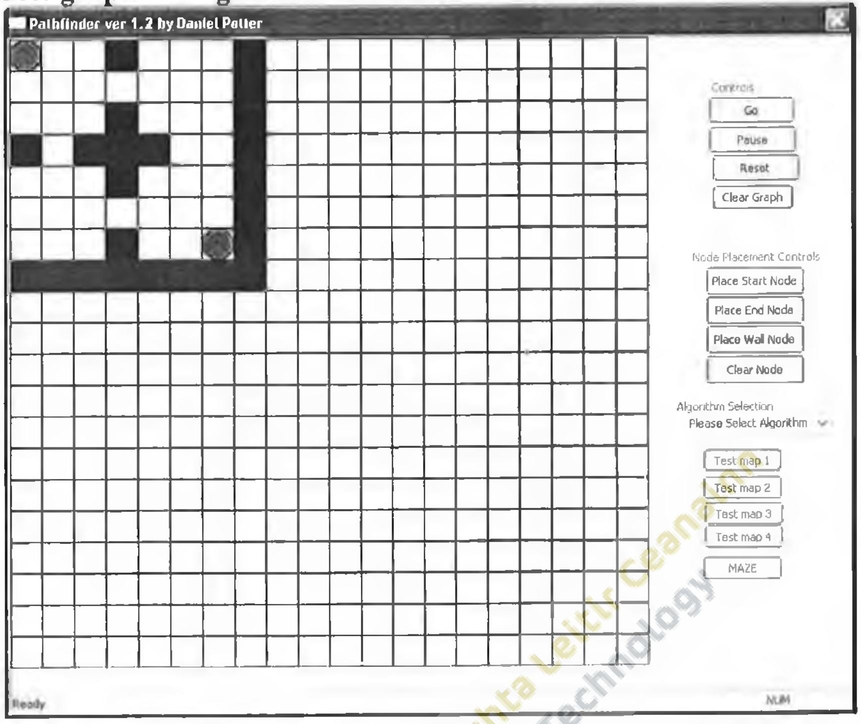


Fig 4.2 Test graph 11 x 11 grid

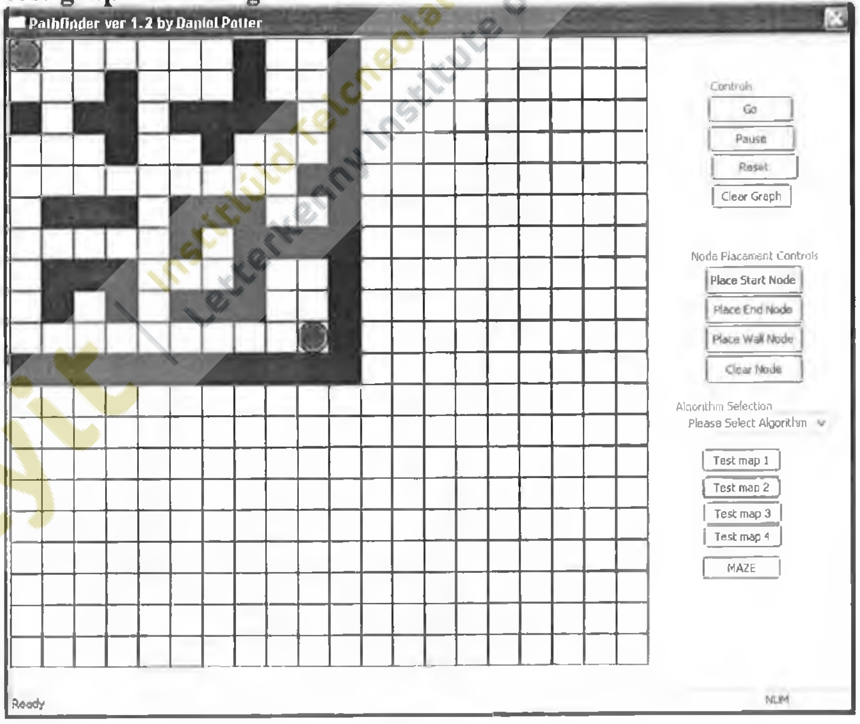


Fig 4.3 Test graph 15 x 15 grid

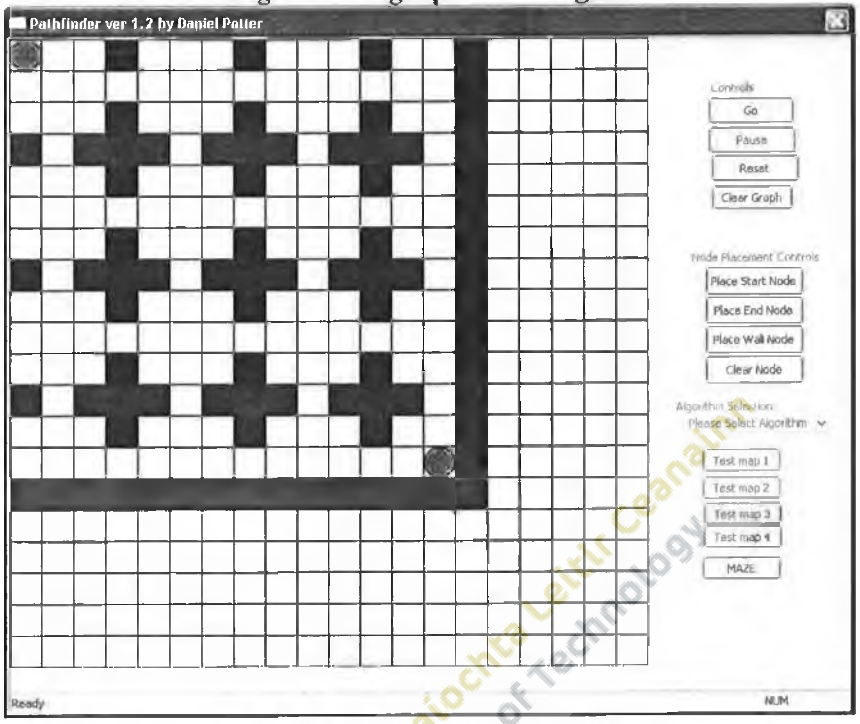
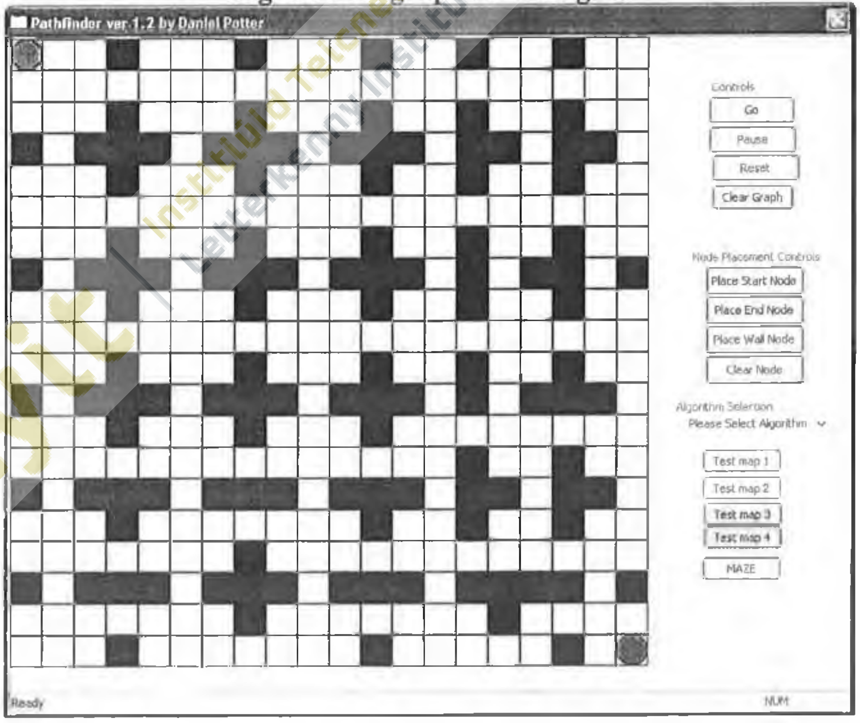


Fig 4.4 Test graph 20 x 20 grid



Once each test has been completed, the data collected can be used to create a graph to show problem size and time completed.

For this research, not all the algorithms documented will be tested; this is due to the fact that some algorithms by design will not be comparable to others. As such the following algorithms will be used in the comparative assessment.

- Breadth first search
- Best first search
- Dijkstra
- A*
- D*
- Pathjoiner

None of the uninformed algorithms will be assessed. For the dynamic algorithms the test will have to be modified due to the difference between these and the traditional pathfinding algorithms. This topic is discussed in Section 5.8. Depth first search is also amiss due to that the fact it wasn't implemented in the application.

When testing was done with the algorithms, it was discovered that the timer implemented would not measure a low enough time scale to record a measurement. To remedy this, a delay (one millisecond) was placed into each algorithm so that a more meaningful measurement of time would be recorded. Upon the completion of processing, the delay would be removed to give us a time value.

All of the algorithms where tested on machine specification given below.

- Intel ® Core ™ 2 CPU 6300 @ 1.86GHz (2 CPUs)
- 1024 MB RAM
- Microsoft Windows XP professional

4.4 Dynamic algorithm analysis

The tests that were outlined in the previous section will not work with dynamic algorithms since they do not take into account that the environment is dynamic, and for the algorithm to fulfil its purpose, change has to take place within the graph. Therefore to generate some meaningful output for dynamic algorithms, new tests where devised which take into account each algorithm. The tests are similar to the previous tests in that they take place on gradually larger graphs. However due to the algorithms being dynamic, two searches will need to take place in order to derive a result.

1. The initial search of the graph before any change has taken place
2. The search of the graph after a change has taken place

For each search the time will be recorded and graphs generated as in previous test examples.
Below are the problem graphs that will be used along with the node change that will take place.

Fig 4.5 Graph 1 before change

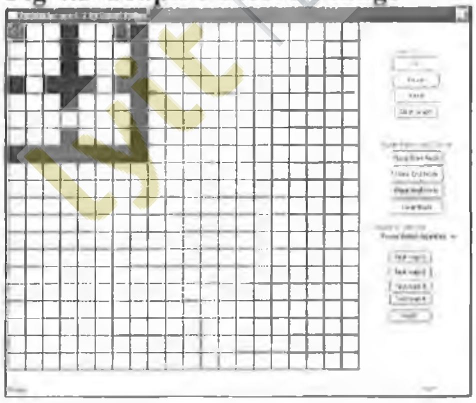


Fig 4.6 Graph 1 after change at 4, 1

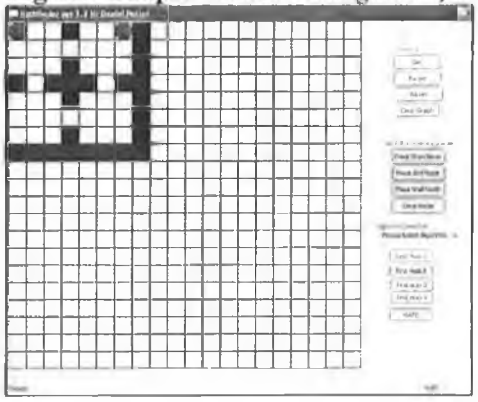


Fig 4.7 Graph 2 before change

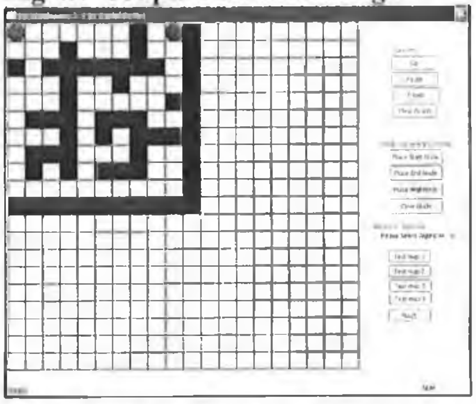


Fig 4.8 Graph 2 after change at 4, 5

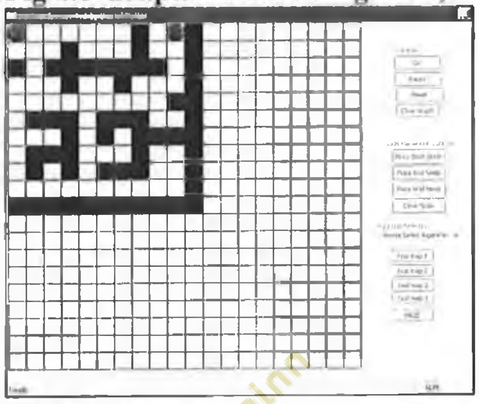


Fig 4.9 Graph 3 before change

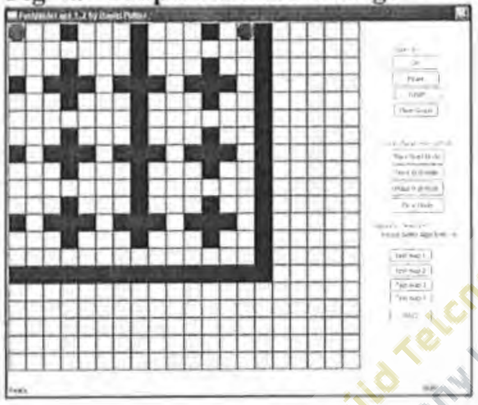


Fig 4.10 Graph 3 after change at 8, 6

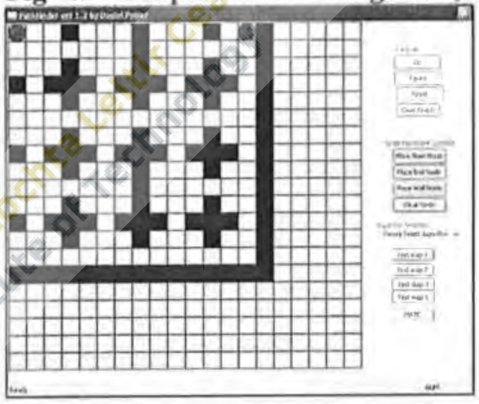


Fig 4.11 Graph 3 before change

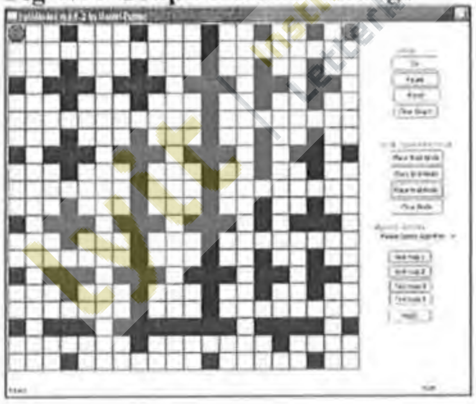
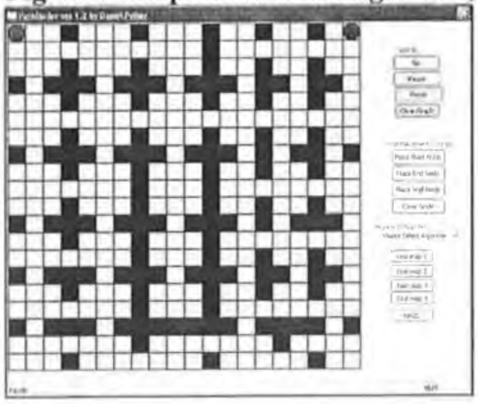


Fig 4.12 Graph 3 after change at 12, 6



4.5 Test results

The following section details each algorithm's performance over the tests which apply to it. All the documented algorithms performed successfully and as optimal as the development allowed them to be.

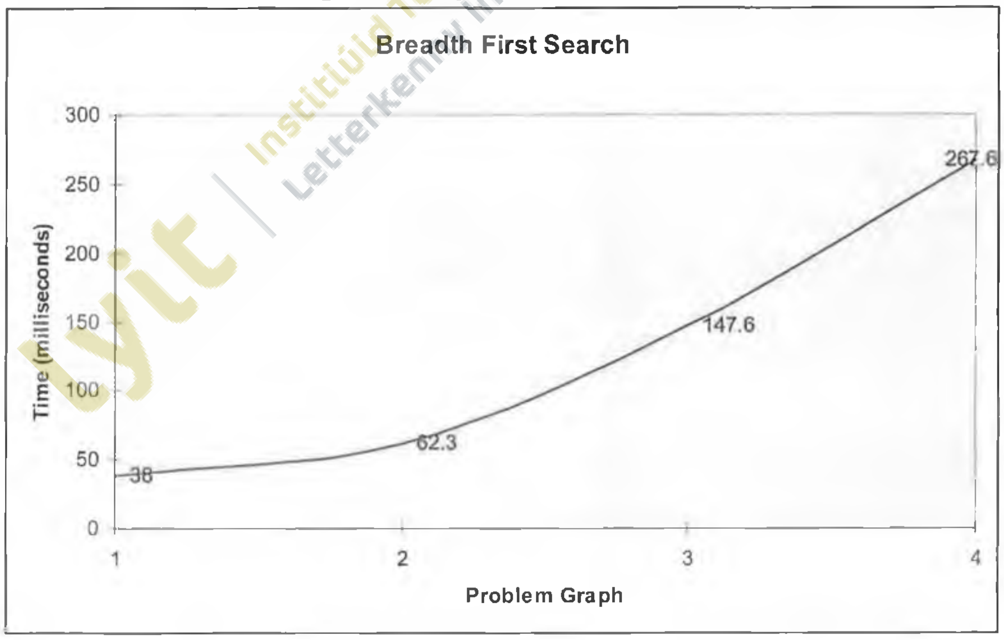
4.5.1 Breadth first search

The following data was generated for the algorithm performance for each problem graph as shown in section 2.7. All the times are measured in milliseconds.

Fig 4.13 Data gathered from pathfinder application for the BFS algorithm

	Graph 1	Graph 2	Graph 3	Graph 4
Test 1 time	38	57	137	267
Test 2 time	38	73	137	267
Test 3 time	38	58	169	269
Average time	38	62.3	147.6	267.6

Fig 4.14 Graph of data from fig 4.13



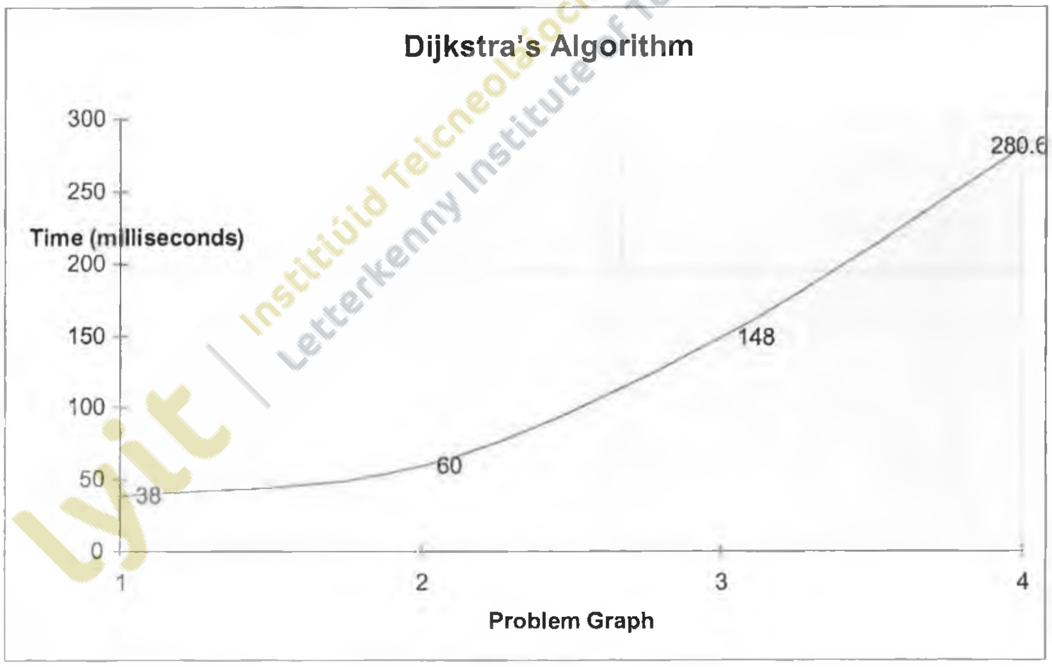
4.5.2 Dijkstra's algorithm

The following data was generated for the algorithm performance for each problem graph as shown in section 2.7. All the times are measured in milliseconds.

Fig 4.15 Data gathered from pathfinder application for Dijkstra's algorithm

	Graph 1	Graph 2	Graph 3	Graph 4
Test 1 time	38	60	153	281
Test 2 time	38	60	153	280
Test 3 time	38	60	138	281
Average time	38	60	148	280.6

Fig 4.16 graph generated from fig 4.15



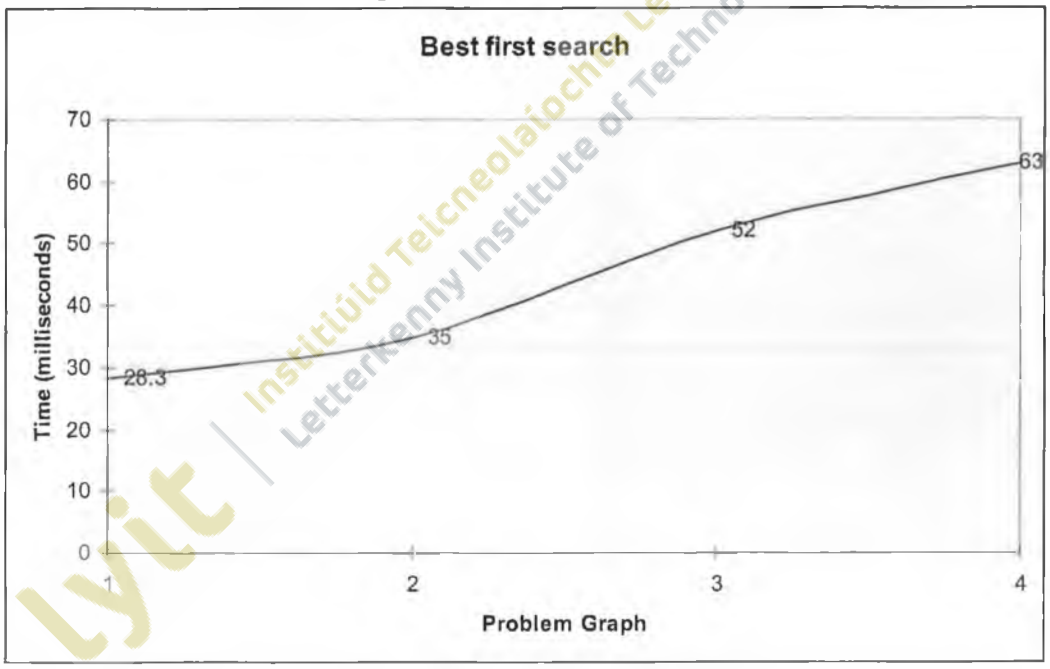
4.5.3 Best first search

The following data was generated for the algorithm performance for each problem graph as shown in section 2.7. All the times are measured in milliseconds.

Fig 4.17 data derived from the processing of the best first search algorithm

	Graph 1	Graph 2	Graph 3	Graph 4
Test 1 time	39	30	63	73
Test 2 time	23	45	47	58
Test 3 time	23	30	46	58
Average time	28.3	35	52	63

Fig 4.18 Graph derived from the data in fig 4.17



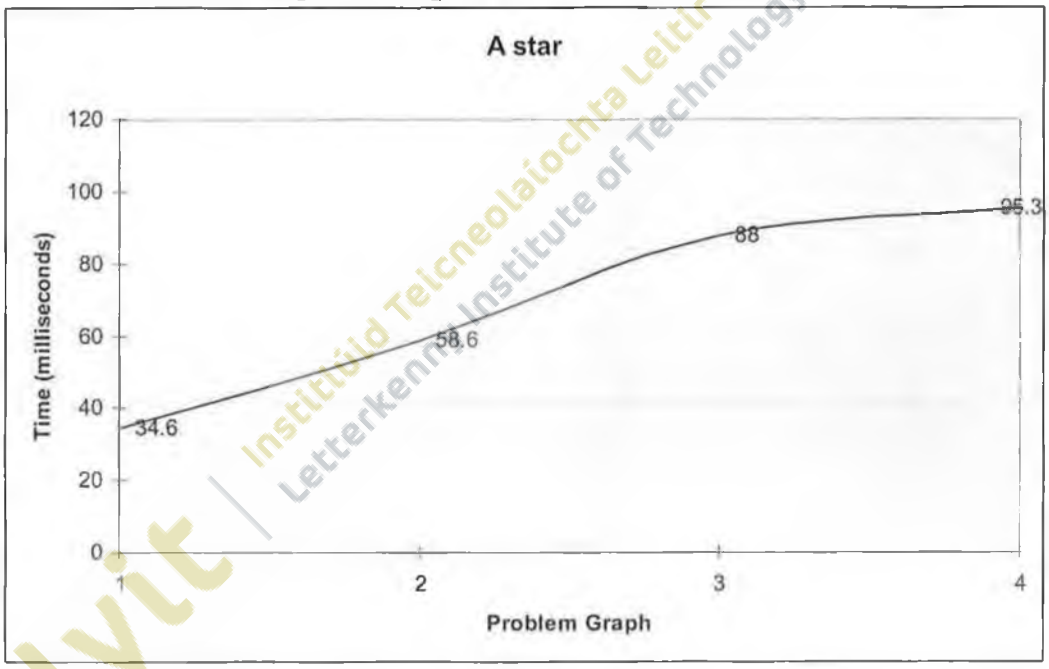
4.5.4 A-star (A*)

The following data was generated for the algorithm performance for each problem graph as shown in section 2.7. All the times are measured in milliseconds.

Fig 4.19 Data derived from the processing of the A* algorithm

	Graph 1	Graph 2	Graph 3	Graph 4
Test 1 time	40	64	88	95
Test 2 time	40	64	88	96
Test 3 time	24	48	88	95
Average time	34.6	58.6	88	95.3

Fig 4.20 Graph of the data from fig 4.19



4.5.5 Dynamic A-star (D*)

Since the tests used to assess the performance of D*are different to that of the static algorithms, two sets of results are applicable to this algorithm's. The run time before any change and the run time to assess the change made.

Fig 4.21 Data gathered for the first search using the D* algorithm

	Graph 1	Graph 2	Graph 3	Graph 4
Test 1 time	17	19	78	148
Test 2 time	17	34	79	149
Test 3 time	17	35	63	149
Average time	17	29.3	73.3	148.6

Fig 4.22 Data gathered for the second search using the D* algorithm

	Graph 1	Graph 2	Graph 3	Graph 4
Test 1 time	37	61	138	268
Test 2 time	53	61	138	270
Test 3 time	37	61	122	270
Average time	42.3	61	132.6	268.6

Fig 4.23 Graph generated using the data from fig 4.21

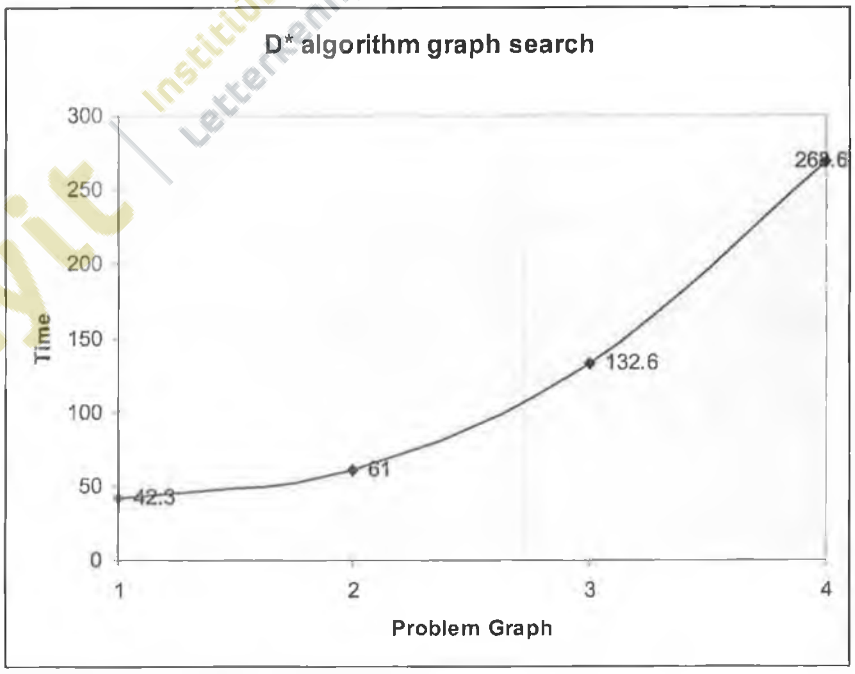
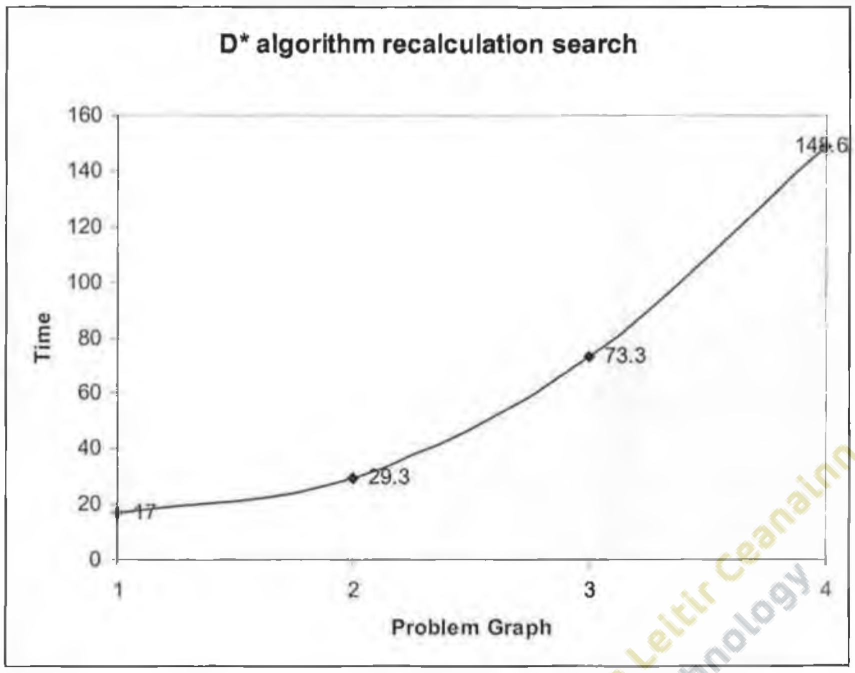


Fig 4.24 Graph generated using the data from fig 4.22



4.6 Result Analysis

As expected with all the algorithms the run time increases as the size and complexity of the problem increases. However with the majority of the tests, run time increases at very noticeable rate as the problem size increases. The exception to this is the A* and BFS algorithms, their increase in run time almost forms a linear pattern on their respective plotted graphs. Using Big O notation, this would indicate that they where O(N), the curve generated by the other algorithms indicate they are quadratic and hence O(N^2).

4.7 References

- Data structures for games programmers, Ron Penton, 2003, ISBN 1931841942, Premier press.
- http://www.microsoft.com/technet/sysinternals/SystemInformation/ClockRes.mspx, Measuring system time resolution, Mark Russinovich. Accessed 2/11/07.

5. Developing a dynamic pathfinding algorithm

5.1 Introduction

All of the pathfinding algorithms (static and dynamic) hitherto considered are relatively common in the field of computer games. Despite the fact the author programmed D* from a textual description there has yet been no attempt made to modify the algorithm to better suit dynamic changes to the graph and that is the focus of this chapter.

5.2 Algorithm design

D* offers the most likely route to solving the problem of dynamic pathfinding to date. For a new algorithm to be implemented, it was useful to look at the flaws of D* and try to remedy them.

In section 2.31 within the conclusions the following insights on D* are described.

1. Each A.I agent using D* for navigation would have to store its own version of the graph so as to avoid conflicts with other A.I agents.
2. D* doesn't create a path, it creates a graph of nodes all of which have to be recalculated in the event of a change.
3. D* requires the processing of the entire graph so that future algorithm recalculation of nodes is performed correctly.

The first point outlines the fact that any A.I. agent using D* will have to store a separate version of the graph for this agent. With standard pathfinding algorithms such as breadth first search and A* (section 2), the agent only needs to store the path generated by the pathfinding algorithm. This is one of the requirements of the new pathfinding algorithm:

1. When the algorithm has completed processing, the output must be a path similar to that generated by static algorithms e.g. A*.

This requirement covers the second point with D*, instead of the graph of signposts it creates a path.

The third point revolves around D* processing of the entire graph, which as stated before is very costly if the graph is extremely large. Ideally an algorithm should only require the processing of nodes that are potentially affected by any change in the graph, any other processing of nodes outside of this affected area could be considered a waste of resources. As such a second requirement of the algorithm would be:

2. Algorithm must restrict processing of nodes to those that have been affected by any change that may happen in the graph.

Finally D* creates and edits its graph all within one algorithm, it could be said that the input for D* is the graph that itself previously created and the output is the edited graph. If this new algorithm that is being designed is to output a path as stated in the first requirement, then its input would be a similar kind of path to what it creates.

3. The algorithm will take as input a standard path and edit it.

This input path, if possible, could be generated by any of the non dynamic pathfinding algorithms. As such it could be possible for the new algorithm to be a path editing algorithm rather than a path generating algorithm.

D* provided useful insights as to how dynamic pathfinding can be achieved. However it has several problems that preclude its further use. Any aspirations of modifying this algorithm to respect the previously stated issues thrown out. Therefore the new algorithm would be built from the ground up, using features of static pathfinding algorithms.

While these three conditions must be satisfied in order to advance from the capabilities of D*, there are further conditions that must also be met in order to have a successful dynamic pathfinding solution. These relate to the modification of a given path.

5.3 Redesigning a given path

With the conclusion made not to modify D* and with the insights into the algorithm clearly pushing for the development of an algorithm which as input takes in a path and a change and as output gives a new path should it be required, rather than create an entirely new path from scratch when a change is made (this can already be done using a static PFA), the choice was made to edit the path that has been taken as input, rather than create a new path. This is so that the edited path reflects any change that has been made in the graph without the necessity of creating a totally new path.

"*Looking for a good route for moving an entity from here to there*"

(Stout, 1997)

The above quote was previously used in section 2.2 to describe pathfinding, in the context of this section it is the word "*route*" that is the focus. The "*route*" is the path generated by a pathfinding algorithm and in the domain of the node graph that a pathfinding algorithm searches, the path is a list of nodes that an AI agent follows in order to reach the point that the pathfinding algorithm was searching for. From this it is determined that the path is a list of nodes, and it is this list that is to be edited.

Fig 5.1 Stored path

PATH LIST
Node C, 1
Node B, 2
Node A, 3
Node B, 4
Node C, 5

Fig 5.2 Visual stored path

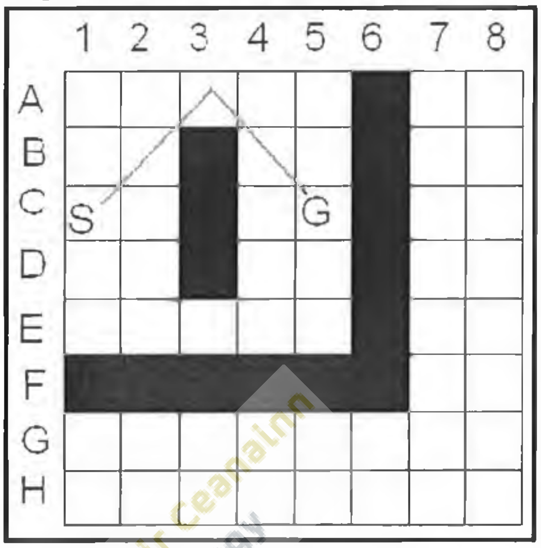


Fig 5.1 shows a list of nodes that are on the path and **Fig 5.2** shows its corresponding path on the graph. Should some change happen in the graph and via the use of an algorithm a change is detected and a better path is found, the list of nodes will need to be edited or recreated so that any AI agent can follow the new path.

The list of nodes on the shortest path is generated using the function that draws the shortest path to the graph as described in section 3.1.6. As the function draws the shortest path to the screen it adds each node into a list of nodes that reflects the path generated.

Environmental addition and environmental subtraction will also have an affect on how any path editing is done since they both create unique problems. With environmental addition we only need to recalculate the path should the addition affect one of the nodes on the actual path e.g. (B, 2). At that point it is definite that a new path will have to be created. With environmental subtraction, the algorithm will have to test to see if it can find a new and better path between nodes on the generated path before it actually generates the path itself. So therefore depending upon the type of change made in the graph it will require a different perspective on creating a solution.

In order to make a complete dynamic pathfinding algorithm, the primary focus was placed upon creating an effective environmental subtraction algorithm with a secondary objective of the algorithm being that it can handle environmental subtraction.

The decision was made to develop an algorithm which tries to "join" two nodes on a given path. The join section will be a path and it will be generated with respect to the change made in the graph via the algorithm. Once the join has been completed it needs to be tested to see if this join is a better path than the existing path that occurs between the two nodes. If it is then the join is edited into the path overwriting what was there previously. In view of this two more conditions where generated for the new algorithm.

4. Use a joining technique to create an alternative path.

5. If the alternative path is shorter than the section of the path that is affected by then overwrite the affected path with the alternative path.

Since the principle mechanism of the algorithm is joining two nodes on a given path, it was decided to name the new algorithm **pathjoiner**.

5.4 High level view of the pathjoiner algorithm

At its highest level this algorithm tries to build a path between two points on the already created shortest path, with respect to each of the conditions that have been outlined in this section. This will give a very broad view of how the algorithm works. Fig 5.3 for reference to X, Y ,Z ,X1 ,X2 and X3.

- **The algorithm will take as input a standard path and edit it.**

This is done by using a static pathfinding algorithm (e.g. A*) to create a path. Once the algorithm is activated by a change in the graph it is this path that will be edited.

- **Use a joining technique to create an alternative path.**

The alternative path is constructed by using an algorithm to find nodes that are on the given shortest path. This search will start at the node where the change occurred in **Fig 5.3** this is node Z. It needs to join up at 2 different nodes on the given shortest path (nodes X and Y). These nodes are tested as described shortly, if there is no successful path, and then the search continues on to the next node.

- **Algorithm must restrict processing of nodes to those that have been affected by any change that may happen in the graph.**

The algorithm starts its processing at the node where the change occurred as outlined in the previous condition. This places the search range of the algorithm directly in the area of the graph affected by the change. Also to restrict processing a cut off point is used to limit how large the search range of the algorithm is and when algorithm termination must occur.

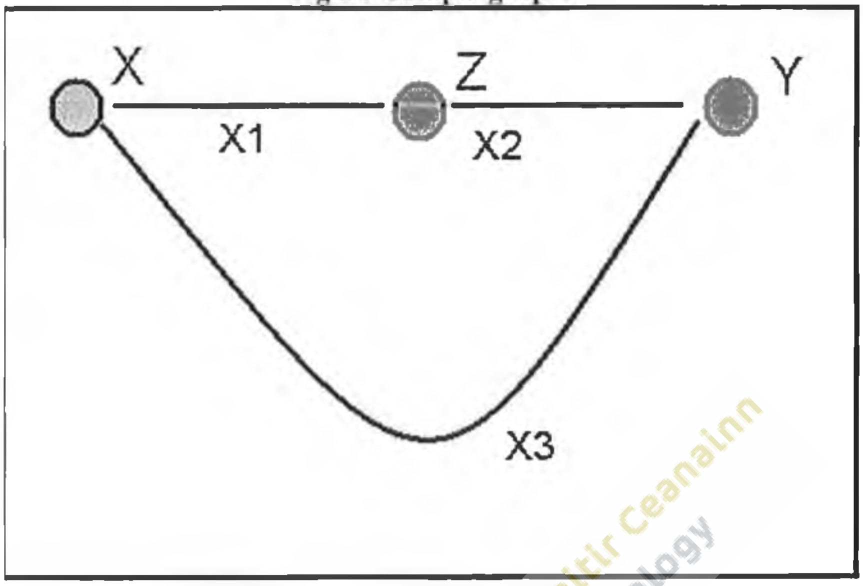
- **If the alternative path is shorter than the section of the path that is affected by then overwrite the affected path with the alternative path.**

To find if this condition is true a formula must be applied to see if an alternative path is shorter than the affected path.
This process is repeated until either the formula becomes true or the cut off point mentioned above comes into affect.

- **When the algorithm has completed processing, the output must be a path shorter than that generated by static algorithms e.g. A*.**

When the formula outline above becomes true then there has been a shorter path found. In order to edit the previous path the parents of nodes where the join has been made have to be edited so that they follow the new path. This new path is then read to memory from the parent's data.

Fig 5.3 Sample graph:



5.5 Algorithm mechanics

With the basics of the pathjoiner algorithm now established, this section will examine the mechanics of the algorithm at a low level with respect to the conditions outlined.

- **The algorithm will take as input a standard path and edit it.**

Fig 5.4 Path Generated

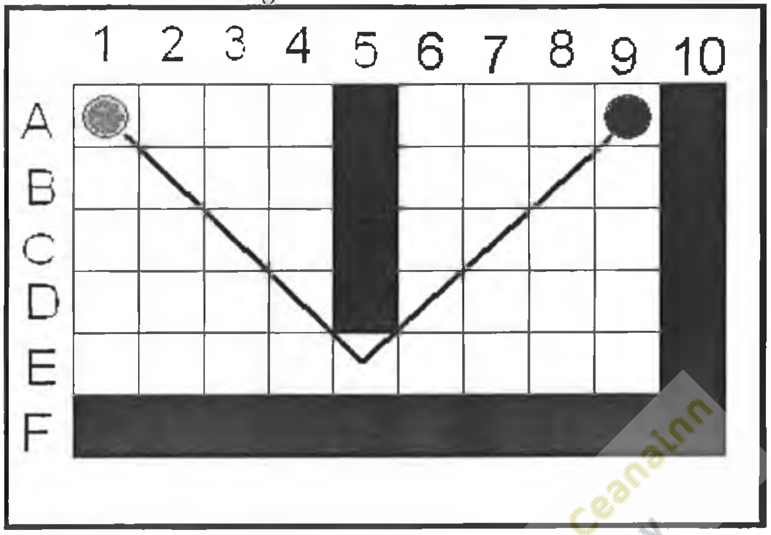


Fig 5.4 shows a graph after the A* pathfinding algorithm has been run upon it creating the shortest path between the start and the end nodes. There is a minor modification made to the node in that it now stores a flag on whether or not it is on the shortest path or not. Also any standard algorithm that can generate a standard path must be able to record the distance from the start node to the current search node, exactly like the Dijkstra algorithm does. At this point there is a change made in the graph, since the algorithm was developed to work with environmental subtraction. One of the nodes that have been assigned as an obstacle has to be cleared.

Fig 5.5 Change made in graph:

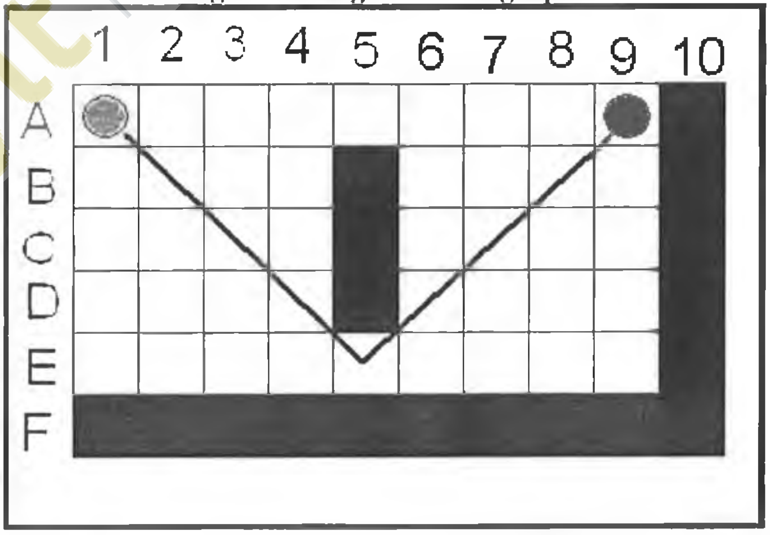


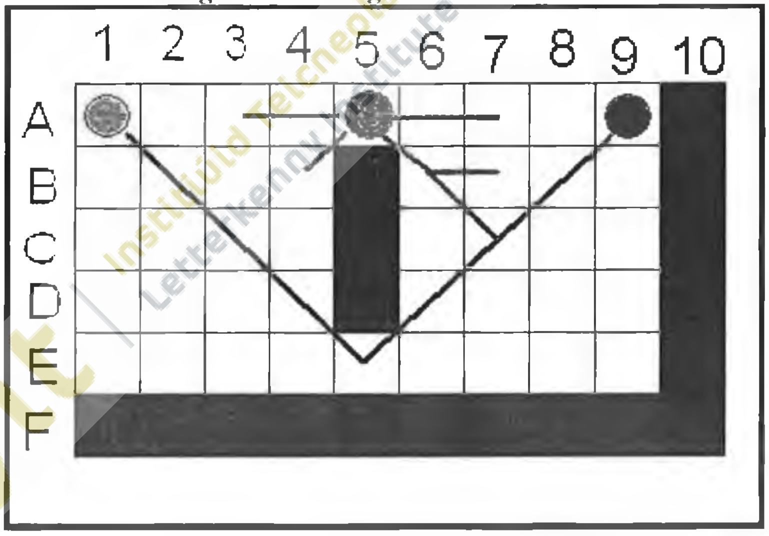
Fig 5.5 shows that the node (A, 5) has been cleared. At this point there has been a change made in the graph and the algorithm must now check to see if it can find a better path.

To restrict the range of the search, it will begin from the node where the subtraction took place. In the example this is node (A, 5) and will be called the source node. The algorithm then commences a modified Dijkstra search from this node.

- **Use a joining technique to create an alternative path.**

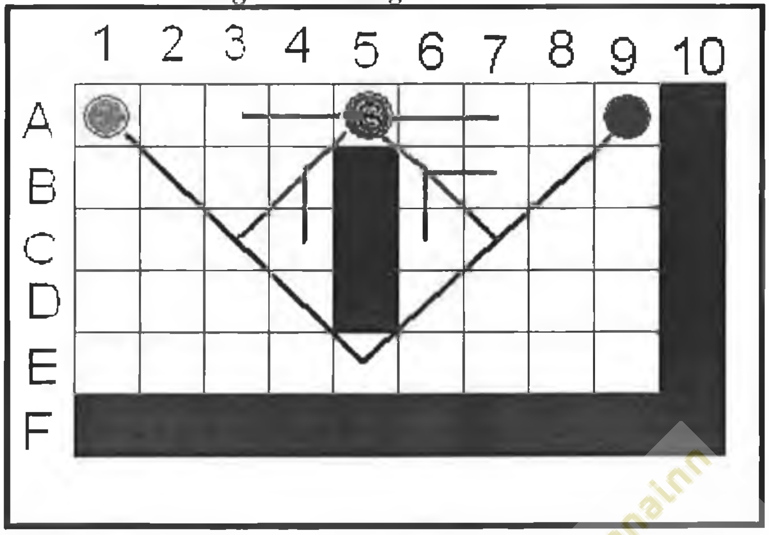
Once the algorithm discovers the first node that is on the shortest path it assigns this node to be the locater node. Due to the way that Dijkstra works as outlined in section 2, this will be the closest node to the source node and in the example this will be node (C, 7).

Fig 5.6 Creating the locater node



The locater node has already a distance assigned to it, this distance is from the original start node. So there needs to be another distance stored that represents the distance of the locater node to the source node. In our example that distance would be 2.8 since it is two diagonal (1.4 distance) moves from the source node. Now that the algorithm has the locater node it can continue with the processing.

Fig 5.7 Finding other nodes



At each search of a node, the algorithm checks to see if this node is on the shortest path. This is achieved using the flag mentioned above instead of searching the list of nodes that are on the shortest path so as to reduce processing.

- **If the alternative path is shorter than the section of the path that is affected then overwrite the affected path with the alternative path.**

When the algorithm finds other nodes on the shortest path it needs to perform operations on that node. Using the example the second node to be found would be node (C, 3) as shown in **Fig 5.7**; its distance would also be 2.8 from the goal node. Since the locater node has already been found, the algorithm can now test to see if the possible path it has generated is better than the existing path. It does this using the following condition.

$$(X1 + X2) < Y$$

- **X1:** This is the distance between the source node and the locater node
- **X2:** This is the distance between the source node and the current node on the shortest path that is being searched.

- **Y:** This is the distance between the locater node and the searching node on the previously generated path.

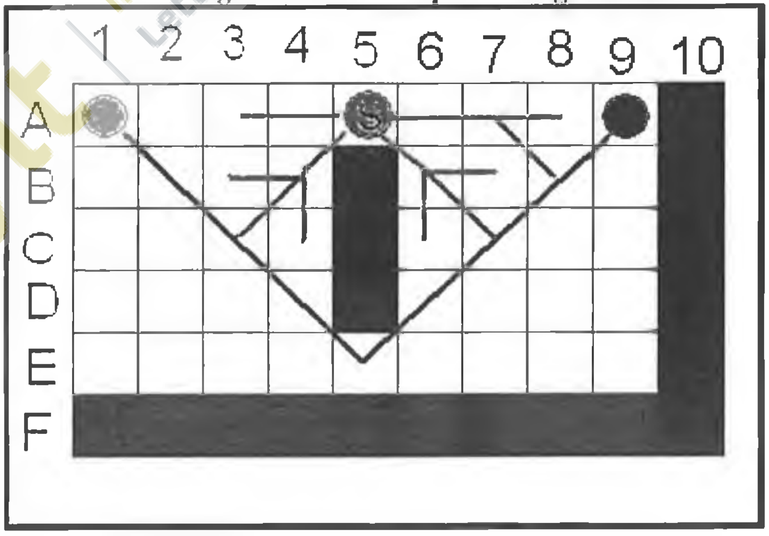
Y is generated by taking the lesser of the two distances between the locater node and the current searched node and subtracting it from the greater. Also of note is if the locater node has a lower distance than that of the current search node, then it is earlier on the path than the current search node. The same principle applies if the current search node has a lower distance.

In our example the locater node (C, 7) is 2.8 distant from the source node and 8.4 from the start node. The current search node (C, 3) is 2.8 distance from the source node and 2.8 from the start node. These are input into the formula and it is exactly the same distance the previous path between the two nodes.

$$\mathbf{(2.8 + 2.8) < (8.4 - 2.8) = 5.6 < 5.6}$$

In this instance there is no point in changing the path if it is the same distance between the two nodes. The algorithm is searching for a lower distance for which to create the path from. Once this is completed, the standard algorithm processing continues and the next node to be found that is on the shortest path is (B, 8) as shown in **Fig 5.8**.

Fig 5.8 Continued processing:

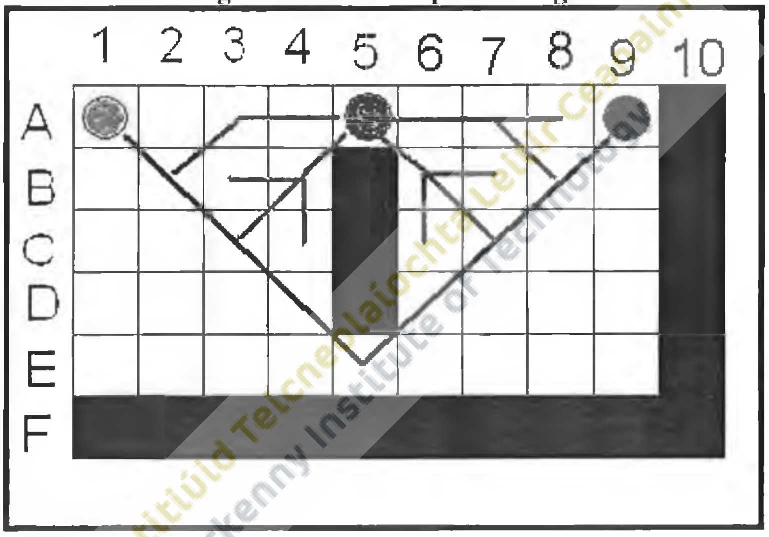


Using the same formula again

$$(2.8 + 3.4) < (9.8 - 8.4) = 6.2 < 1.4$$

In this instance it is clear that the temporary path is not as good as the previously generated path, and the algorithm continues processing normally. And the next node to be found is node (B, 2) as shown in **Fig 5.9**.

Fig 5.9 Continued processing:



Using the same formula stated above

$$(2.8 + 3.4) < (8.4 - 1.4) = 6.2 < 7$$

In this instance it is clear that the temporary generated path is shorter than the previous generated path between the locater node and current search node. Now that the algorithm has generated the temporary route, the node list used to store the previous path needs to be edited so that it takes the new path into account.

- **When the algorithm has completed processing, the output must be a path similar to that generated by static algorithms e.g. A*.**

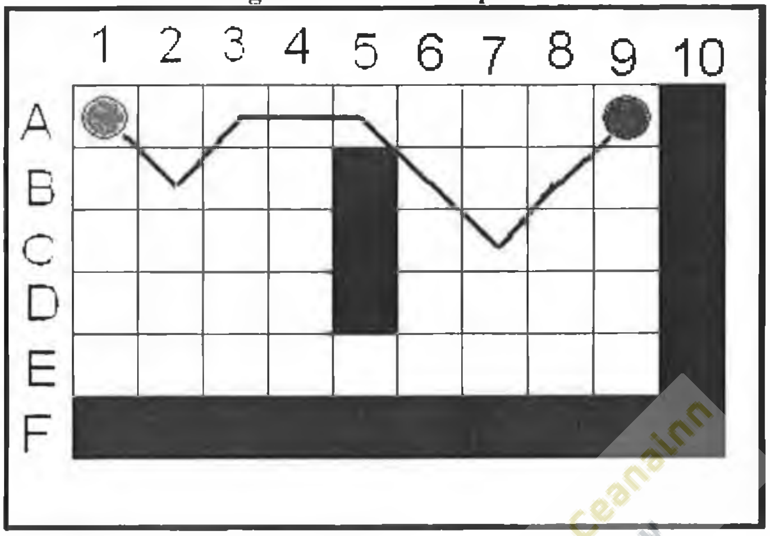
Section 3.1.6 outlines how the application fills the list that stores any generated path; the application can reuse this function to re-create the new list that incorporates the new path. The function that performs this task works by examining the parent nodes of each node beginning with the goal node. Logically following the parent will always lead the path back to the start node. Using this method would mean that the parent nodes of nodes on the path would need to be altered so that it follows the newly created shorter path rather than the previous shorter path.

The process of editing the parent nodes begins with changing the parent node of the node with the greater distance from the original start node where the new path and the previous path connect. The node in question is either the current search node or the locater node; in the example it is the locater node. The original parent node of node (C, 7) is node (D, 6) which is on the previous path. The parent node of (C, 7) is set to node (B, 6), this operation connects the new path to the previous path at this node as far as the source node since the parent of node (A, 4) (the node after the before the source node on the new path) is the source node itself.

Starting from the node that didn't have the longest distance, i.e. the current search node in the example, the algorithm performs several copy operations on the each of the node's parent node until it reaches the source node itself. This operation connects the parent nodes from source node to the current search node.

Once this is completed all the parents are edited correctly and the function to create the list of shortest paths is called. Once completed the path looks like the following one shown in **Fig 5.10**.

Fig 5.10 The edited path:



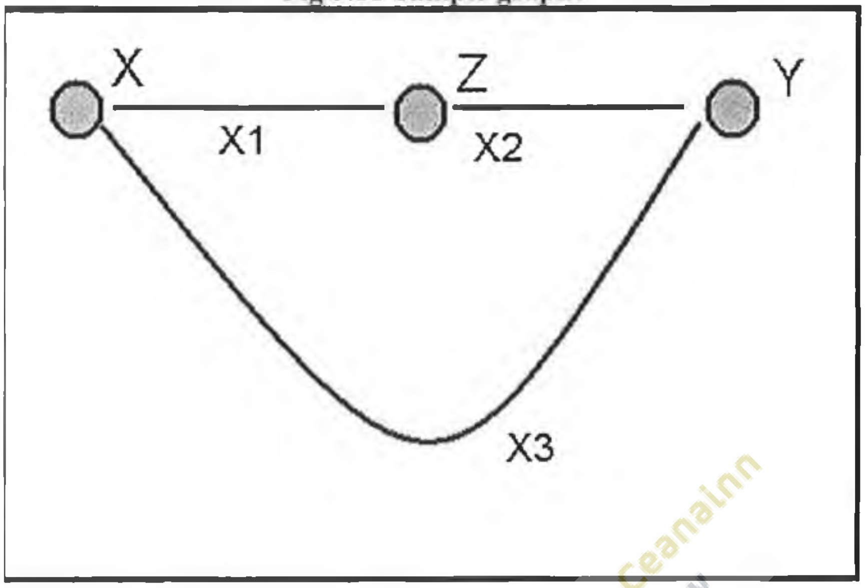
- **Algorithm must restrict processing of nodes to those that have been affected by any change that may happen in the graph.**

The cut off point for any pathfinding algorithm is when the algorithm stops processing. The cut off for standard pathfinding algorithms (e.g. A-star) will only happen in one of three occasions.

- Algorithm succeeds and finds a path
- No more nodes to process
- Algorithm specific cut off

D* will finish processing when it has no more nodes to process, meaning that it has to go through the entire graph before it ceases. As mentioned before this is waste of resources.

Fig 5.11 Sample graph:



For this new algorithm it will cut off when there are no more nodes to process similar to that of D*. In order not to process the entire graph like D* a secondary cut off point is introduced. Using **Fig 5.11**, if X was the start node and Y was the end node. And the total distance of the shortest path was X3. If at any point the combined distance of the two points (X1 + X2) from the source node is greater than the length of the entire shortest path then it is guaranteed that there is no shorter path to be found.

This is due to how the Dijkstra algorithm that powers the search works. As mentioned in section 2.18, Dijkstra works by keeping track of the distance from the search node to the start node. In the search algorithm once the distance of the temporary path generated exceeds the total distance of the generated shortest path then it is guaranteed that there can't be shorter path generated.

Using this extra cut off point will reduce the excess processing of nodes within the search graph.

The author suggests that this cut off point can be further reduced by half. This could be done by checking the distance during the creation of the source node. If at any point this is greater than half of the distance of the final shortest path, then it should be guaranteed that once a source node is created, the total distance of the source node and the locater node will be guaranteed to be greater than the total distance of final shortest path. This is because the distance of the locater node will be, at minimum the distance of the source node. This is not implemented in the pathfinder application developed as part of this research.

5.6 Critical analysis

Using the same performance measures as defined in section 4.1.3 of this document the algorithm was analysed to see how effective it is at solving the dynamic pathfinding problem.

- **Accuracy:** The first and most obvious feature of the algorithm as shown in the final path generated in the above example as shown in **Fig 6.9** is not guaranteed to find the shortest path. This lack of accuracy is due to the Dijkstra search from the source node, Dijkstra will find the shortest points on the previous shortest path to the search node. These points will be the shortest points in order to the source node, but it is not guaranteed that these points will generate the shortest path. Depending upon the type of pathfinding problem that this algorithm tries to solve, the accuracy will vary.

- **Efficiency:** The efficiency of the algorithm was determined by how much the second requirement was adhered to.
  2. Algorithm must restrict processing of nodes to those that have been affected by any change that may happen in the graph.

  Since the algorithm is based upon Dijkstra it shares its problems with unnecessary processing, although this factor is reduced through two features

  - **Algorithm processing starts where the change happens in the graph.** This guarantees that the search area in the graph processed by the algorithm will be the affected area of the graph.

- **The enforcement of a cut off point**. While the first point guarantees the problem area is processed, the cut off point guarantees that it will not do any processing outside of the problem area.

The efficiency of the algorithm will be determined as being quite efficient.

- **Run time:** The run time is determined by algorithm complexity, accuracy and efficiency. Again since the algorithm is based upon Dijkstra it shares its simplicity. When the modifications made to Dijkstra are taken into consideration there aren't any additional features that require any major processing. Therefore the final developed algorithm will be considered as not being complex.
  When this factor is added to the efficiency and the accuracy it is determined that the run time would depend on the type of problem encountered, as it could range from low to high.

To draw a comparison on how well the pathjoiner algorithm compares against D*, a graph was drawn up using the same performance measures as defined in section 4.2.

The same tests in section 4.4 that where used to find out the efficiency of D* will be used to assess the effectiveness of the pathjoiner algorithm. However due to the fact pathjoiner's only function is to recalculate a path due to a change in the graph, the only data to be gathered is from when a recalculation using pathjoiner is performed. The algorithm used to generate the path used by pathfinder was A*. All the times below are measured in milliseconds.

Fig 5.12 Recalculation times for each problem graph using pathjoiner

	Graph 1	Graph 2	Graph 3	Graph 4
Test 1 time	6	7	15	76
Test 2 time	6	7	15	76
Test 3 time	5	7	15	76
Average time	5.6	7	15	76

Fig 5.13 D*'s recalculation times for the same problems

	Graph 1	Graph 2	Graph 3	Graph 4
Test 1 time	37	61	138	268
Test 2 time	53	61	138	270
Test 3 time	37	61	122	270
Average time	42.3	61	132.6	268.6

Fig 5.14 Graph generated from data in Fig 5.12

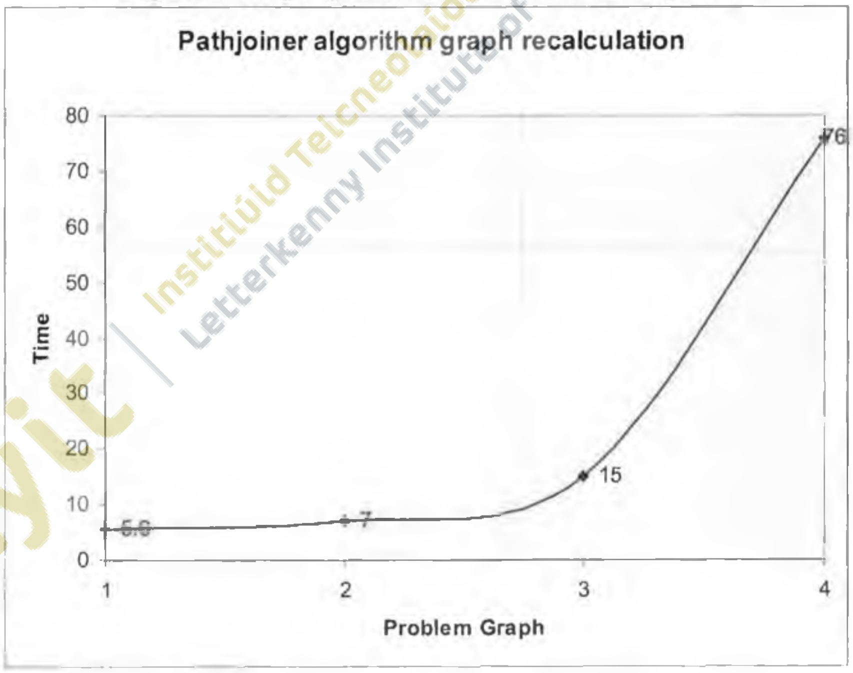
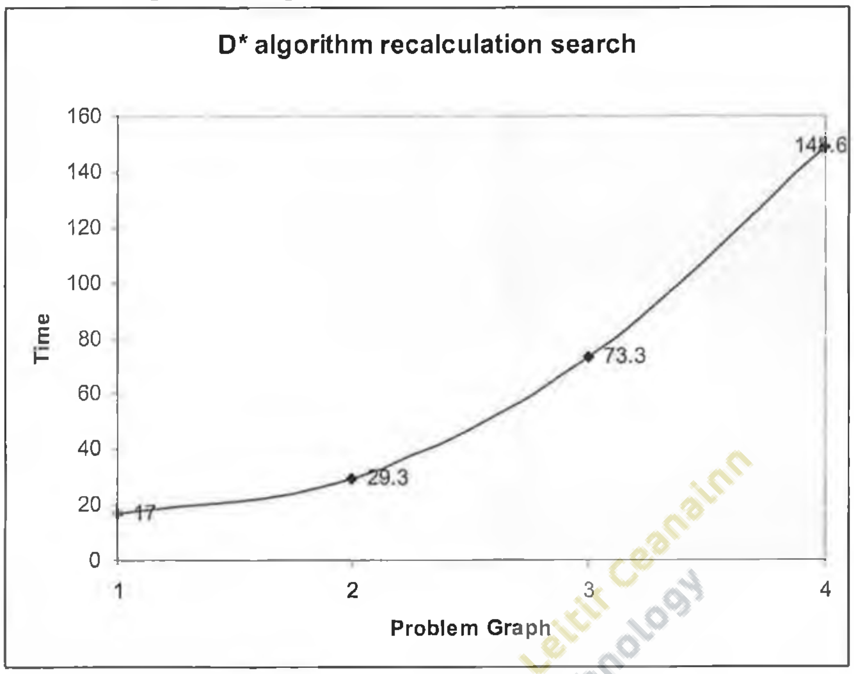


Fig 5.15 Graph generated from data in Fig 5.13



5.7 Conclusions

At the conclusion of this chapter the pathjoiner algorithm has been designed, documented and implemented successfully. The algorithm isn't totally new technology; it is a heavily modified Dijkstra search that has been adapted to solving dynamic pathfinding problems. The implementation in the pathfinder application is a success; it works as the description in this chapter details.

From **Fig 5.14** it is obvious that the bigger the problem, the larger amount of time it takes to perform the search for an alternate path. This is due to the Dijkstra heritage of the algorithm. But for small recalculations it is very quick.

Although the algorithm has its problems, and these are not addressed within this research the following outlines possible further threads of research that could be done.

The algorithm developed deals only with graph subtraction. It seems plausible that the algorithm can be modified to handle addition, but the time constraints of this research preclude any further study in this area.

If one was to pursue this research in this direction it may be useful to possibly adapt the algorithm to the graph subtraction by making the source node and the start node the same as where the change severs the path. Then using the search for the locater node to find any node that occurred after the severance of the path. The same path modifying technique can be used to modify the path to incorporate the change.

The algorithm is not returning the shortest path possible in many instances. Again, time restraints have cut short the research of a solution to this problem.
A possible solution may be found by making the search for the source node dynamic; this will add to the complexity of the algorithm but it will make it more accurate. Also making the locater node search more intelligent by not accepting the first node it comes across that meets the conditions of the formula may help.

6. Conclusions

6.1 Original research concept

The original concept of this research was to document pathfinding algorithms used in digital computer games, with the aim being to test each of the algorithms performance with regard to run time and accuracy. Having completed this, the research aim was turned to investigate dynamic pathfinding and any algorithms which solve the dynamic pathfinding problem. Finally to further back up the research, a unique dynamic pathfinding algorithm was developed to remedy the problems associated with D*.

6.2 Artificial Intelligence in digital games

Artificial Intelligence plays a core part in most modern digital games. Modern players not only want to have a challenging AI to compete against, but they also want an AI which is believable. With the extra processing power afforded by modern computers, more complex AI agents can be developed without having a detrimental effect on the game quality.

6.3 Non Dynamic Pathfinding algorithms

From this research, it has been determined that A* is the best pathfinding algorithm for the general search. The testing performed in section 4, demonstrates that the Best First Search and A* algorithms have consistent level of efficiency over different size problems. It is also been documented that the best first search algorithm is by design more efficient than A* at the expense of the accuracy of the path generated. This leads us into an analysis of the efficiency of each algorithm. This is turn generates an accuracy versus performance debate on which algorithm is to be deemed the most efficient at solving a pathfinding problem in terms of accuracy and performance. The author of this document believes that the trade off to get the most accurate path via A* is more important than the algorithm having a small run time.

However this doesn't restrict developers to strictly using the A* algorithm. Depending upon the search to be performed other PFAs would be of more use than A*. Best first search is more efficient and has similar accuracy to A* over short simple searches.

The most redundant of the algorithms tested would be breadth first search. It could be argued that due to the test results in section 4, that the Dijkstra algorithm was the least optimal. However A* and the later D* algorithms are built using parts of the Dijkstra algorithm and the author believes that to further develop PFAs, knowledge of this algorithm is necessary.

6.4 Dynamic Pathfinding Algorithms

At the time of writing, the number of games that require dynamic pathfinding is increasing. Modifiable gaming environments are no longer a feature for specific games; they have become a standard which the player expects. As a result more games are requiring some kind of dynamic pathfinding solution.

This document concentrates on one particular dynamic pathfinding algorithm (D*, page 64). The tests performed on the algorithm in chapter 4 show that D* efficiency greatly decreases depending upon the size of the problem to be solved. This fact is reflected upon the initial graph processing and reprocessing due to a change.

If we compare the results of the graph generated from Dijkstra's algorithm (**Fig 4.16**) and the results from the initial search of D* (**Fig 4.23**), we can see that the graphs are virtually the same in respect to visual appearance and run time. This shows D*'s dependence on its origins from the Dijkstra algorithm.

The author believes that for dynamic pathfinding to be performed both efficiently and accurately, developers need to look beyond D* and come up with their own solutions. D* lays down the foundations to solving the problem, but it doesn't solve it to the high standard which is expected.

6.5 Pathjoiner algorithm

The pathjoiner algorithm was developed to try and solve some of the problems associated with existing dynamic pathfinding algorithms. The algorithm uses a path generated by an efficient pathfinding algorithm and edits accordingly to any change that may occur in the graph. Straight away this has an advantage over existing dynamic pathfinding algorithms since that it doesn't require the heavy processing cost of initial searching.

A direct comparison between pathjoiner's and D* recalculation is displayed in **Fig 5.6** and **Fig 5.7**. For all problem instances it is clear that pathjoiner is more efficient at solving the dynamic problem than D*. However for the final problem search, the run time for the pathjoiner algorithm grows at a greatly increased rate, being 5 times longer than the previous problems time. However, this time is still quicker than that of D*s for the same problem, and could be considered a successful alternative to D*.

The algorithm demonstrated here is only a test version, given time the algorithm could be developed further. The author believes that there are further modifications that could be made to the algorithm that could make it more efficient and more accurate.

6.6 Pathfinder application

The development of the application was useful to reinforce the author's knowledge of the algorithms. It is also useful for any reader of this document as a visual demonstration of how each of the algorithms work over a user defined search domain. It also provides statistics at the end of each search so that the user can gather an understanding as to which algorithm has the best performance.

6.7 Future research

The main recommendation for further research is the improvement of existing dynamic pathfinding algorithms. This research may not entirely focus upon the algorithms themselves but could be focused upon making the search domain more efficient for dynamic pathfinding algorithms. This research could lead to an entirely dynamic pathfinding solution where algorithms which work in a static search domain such as A* are made totally redundant.

Bibliography

- Steven M. Woodcock, http://www.gameai.com/ ,accessed 2/10/05, Last Update 7/10/05.
- Rules of Play, Eric Zimmermann and Katie Salen, October 2003, ISBN 1556227353, the MIT press.
- Game Design Theory and Practice, Richard Rouse, February 2001, ISBN 1556227353, Wordware Publishing Inc.
- Game Level Design, Ed Byrne, 2005, ISBN 1-58450-369-6, Charles River Media.
- Game Architecture and Design, Andrew Rollings & Dave Morris, 2000, ISBN 1-57610-425-7, the Coriolis Group.
- Introduction to 3D Game Engine Design Using DirectX 9 and C#, Lynn T. Harrison, 2003, ISBN 1-59059-081-3, Apress.
- Windows Game Programming for Dummies 1st Edition, Andre LeMothe, 1998, ISBN 0764503375, Hungry Minds.
- AI for game developers, David M Bourg, O'Reilly, 2004, ISBN 0596005555.
- http://www.gamasutra.com/features/19970801/pathfinding.htm, Smart moves: Intelligent Pathfinding, Accessed 19/10/05, Last Update August 1997, Bryan Stout, Article appeared in "Games developer" magazine October 1996.
- http://en.wikipedia.org/wiki/Search_algorithm, Search Algorithm, Accessed 27/10/05, Last Update 26/10/05.
- Artificial Intelligence structures and strategies for complex problem solving 4th edition, George F Luger, Addison Wesley, 2001, 0-201-64866-0.
- http://sc.tri-bit.com/Dijkstra's_Shortest_Path, Djikstra's shortest Path, Accessed 27/10/05, last update 15/12/04.
- http://www.counter-strike.net, Counter Strike Source Official Website, Accessed 3/3/06.
- AI for Game Developers, David M. Bourg & Glenn Seemann, First Edition July 2004, ISBN: 0-596-00555-5.
- The Complete Brothers' Grimm Fairy Tales, Brothers Grimm, May 1993
- http://www.microsoft.com/technet/sysinternals/SystemInformation/ClockRes.mspx, Measuring system time resolution, Mark Russinovich. Accessed 2/11/07.

- http://collective.valve-erc.com/index.php?go=hammer, Valve hammer level editor, Accessed 14/12/07.
- A* pathfinding for beginners, http://www.policyalmanac.org/games/aStarTutorial.htm, Patrick Lester, Accessed 15/11/05, Last Update 18/7/05.
- Games Programming Gems 5, Kim Pallister, “Beyond A*” by Mario Grimani and Mathew Titilbaum, 1-58450-352-1, Charles River Media, 2005.
- Data structures for games programmers, Ron Penton, 2003, Premier press.
- http://www.microsoft.com/technet/sysinternals/SystemInformation/ClockRes.mspx, Measuring system time resolution, Mark Russinovich. Accessed 2/11/07.

Appendices

Code for pathfinder Application

Pathfinder.h

```
/////////////////////////////////////////////////////////////////////////////
// pathfinder.h : main header file for the pathfinder application
/////////////////////////////////////////////////////////////////////////////
#pragma once

#ifndef __AFXWIN_H__
	#error "include 'stdafx.h' before including this file for PCH"
#endif

#include "resource.h"       // main symbols


// CpathfinderApp:
// See pathfinder.cpp for the implementation of this class
//

class CpathfinderApp : public CWinApp
{
public:
	CpathfinderApp();

// Overrides
public:
	virtual BOOL InitInstance();

// Implementation
	afx_msg void OnAppAbout();
	DECLARE_MESSAGE_MAP()
};

extern CpathfinderApp theApp;
```

Pathfinder.cpp

```
//////////////////////////////////////////////////////////////////////
// pathfinder.cpp : Defines the class behaviors for the application.
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "pathfinder.h"
#include "MainFrm.h"

#include "pathfinderDoc.h"
#include "pathfinderView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CpathfinderApp

BEGIN_MESSAGE_MAP(CpathfinderApp, CWinApp)
	ON_COMMAND(ID_APP_ABOUT, &CpathfinderApp::OnAppAbout)
END_MESSAGE_MAP()


// CpathfinderApp construction
CpathfinderApp::CpathfinderApp()
{
	// TODO: add construction code here,
	// Place all significant initialization in InitInstance
}


// The one and only CpathfinderApp object
CpathfinderApp theApp;

// CpathfinderApp initialization
BOOL CpathfinderApp::InitInstance()
{
	m_pMainWnd = new CMainFrame() ;
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	return TRUE;
}
```

```
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
	CAboutDlg();

// Dialog Data
	enum { IDD = IDD_ABOUTBOX };

protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
}

// App command to run the dialog
void CpathfinderApp::OnAppAbout()
{
	CAboutDlg aboutDlg;
	aboutDlg.DoModal();
}
```

resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by pathfinder.rc
//
#define IDD_ABOUTBOX                    100
#define IDP_OLE_INIT_FAILED             100
#define IDD_TOOLBAR                     103
#define IDR_MAINFRAME                   128
#define IDI_ICON1                       132
#define IDI_ICON2                       133
#define IDC_BTN_GO                      1000
#define IDC_BTN_PAUSE                   1001
#define IDC_COMBO1                      1003
#define IDC_TESTBOX                     1004
#define IDC_TESTBOX2                    1005
#define IDC_BTN_START                   1006
#define IDC_BTN_CLEAR                   1007
#define IDC_BTN_END                     1008
#define IDC_BTN_WALL                    1009
#define IDC_STATUS                      1010
#define IDC_BTN_RESET                   1011
#define IDC_BTN_CLEARGRAPH              1012
#define IDC_BUTTON1                     1014
#define IDC_Load1                       1014

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        133
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1015
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

pathfinder.res

```
// Microsoft Visual C++ generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

/////////////////////////////////////////////////////////////////////////////
//
// Menu
//

IDR_MAINFRAME MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",            ID_FILE_NEW
        MENUITEM "&Open...\tCtrl+O",        ID_FILE_OPEN
        MENUITEM "&Save\tCtrl+S",           ID_FILE_SAVE
        MENUITEM "Save &As...",             ID_FILE_SAVE_AS
        MENUITEM SEPARATOR
        MENUITEM "&Print...\tCtrl+P",       ID_FILE_PRINT
        MENUITEM "Print Pre&view",          ID_FILE_PRINT_PREVIEW
        MENUITEM "P&rint Setup...",         ID_FILE_PRINT_SETUP
        MENUITEM SEPARATOR
        MENUITEM "Recent File",             ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                   ID_APP_EXIT
    END
    POPUP "&Edit"
    BEGIN
    MENUITEM "&Undo\tCtrl+Z",           ID_EDIT_UNDO
    MENUITEM SEPARATOR
    MENUITEM "Cu&t\tCtrl+X",            ID_EDIT_CUT
    MENUITEM "&Copy\tCtrl+C",           ID_EDIT_COPY
    MENUITEM "&Paste\tCtrl+V",          ID_EDIT_PASTE
  END
  POPUP "&View"
  BEGIN
    MENUITEM "&Status Bar",             ID_VIEW_STATUS_BAR
  END
  POPUP "&Help"
  BEGIN
    MENUITEM "&About pathfinder...",    ID_APP_ABOUT
  END
END


/////////////////////////////////////////////////////////////////////////////
//
// Accelerator
//

IDR_MAINFRAME ACCELERATORS
BEGIN
  "N",          ID_FILE_NEW,          VIRTKEY, CONTROL
  "O",          ID_FILE_OPEN,         VIRTKEY, CONTROL
  "S",          ID_FILE_SAVE,         VIRTKEY, CONTROL
  "P",          ID_FILE_PRINT,        VIRTKEY, CONTROL
  "Z",          ID_EDIT_UNDO,         VIRTKEY, CONTROL
  "X",          ID_EDIT_CUT,          VIRTKEY, CONTROL
  "C",          ID_EDIT_COPY,         VIRTKEY, CONTROL
  "V",          ID_EDIT_PASTE,        VIRTKEY, CONTROL
  VK_BACK,      ID_EDIT_UNDO,         VIRTKEY, ALT
  VK_DELETE,    ID_EDIT_CUT,          VIRTKEY, SHIFT
  VK_INSERT,    ID_EDIT_COPY,         VIRTKEY, CONTROL
  VK_INSERT,    ID_EDIT_PASTE,        VIRTKEY, SHIFT
  VK_F6,        ID_NEXT_PANE,         VIRTKEY
  VK_F6,        ID_PREV_PANE,         VIRTKEY, SHIFT
END


/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_ABOUTBOX DIALOGEX 0, 0, 235, 55
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP |
WS_CAPTION | WS_SYSMENU
CAPTION "About pathfinder"
```

```
FONT 8, "MS Shell Dlg", 0, 0, 0x1
BEGIN
   ICON          128,IDC_STATIC,11,17,20,20
   LTEXT          "pathfinder Version 1.0",IDC_STATIC,40,10,119,8,SS_NOPREFIX
   LTEXT          "Copyright (C) 2006",IDC_STATIC,40,25,119,8
   DEFPUSHBUTTON   "OK",IDOK,178,7,50,16,WS_GROUP
END

IDD_TOOLBAR DIALOGEX 0, 0, 121, 326
STYLE DS_SETFONT | DS_3DLOOK | DS_FIXEDSYS | WS_CHILD
FONT 8, "MS Shell Dlg", 0, 0, 0x0
BEGIN
   PUSHBUTTON      "Go",IDC_BTN_GO,33,35,54,16
   PUSHBUTTON      "Pause",IDC_BTN_PAUSE,33,52,55,16
   GROUPBOX       "Controls",IDC_STATIC,29,25,67,87
   COMBOBOX       IDC_COMBO1,17,221,93,17,CBS_DROPDOWN |
WS_VSCROLL | WS_TABSTOP
   EDITTEXT       IDC_TESTBOX,23,252,66,15,ES_AUTOHSCROLL |
ES_READONLY | NOT WS_VISIBLE
   GROUPBOX       "TEST DATA",IDC_STATIC,13,241,87,46,NOT WS_VISIBLE
   EDITTEXT       IDC_TESTBOX2,26,269,59,15,ES_AUTOHSCROLL |
ES_READONLY | NOT WS_VISIBLE
   GROUPBOX       "Algorithm Selection",IDC_STATIC,7,213,102,26
   PUSHBUTTON      "Place Start Node",IDC_BTN_START,31,135,62,17
   PUSHBUTTON      "Clear Node",IDC_BTN_CLEAR,31,187,62,17
   PUSHBUTTON      "Place End Node",IDC_BTN_END,31,152,62,17
   PUSHBUTTON      "Place Wall Node",IDC_BTN_WALL,31,169,62,17
   GROUPBOX       "Node Placement Controls",IDC_STATIC,17,125,85,85
   EDITTEXT       IDC_STATUS,20,304,70,15,ES_AUTOHSCROLL |
ES_READONLY | NOT WS_VISIBLE
   PUSHBUTTON      "Reset",IDC_BTN_RESET,35,69,55,16
   PUSHBUTTON      "Clear Graph",IDC_BTN_CLEARGRAPH,35,87,51,15
   PUSHBUTTON      "Load Map",IDC_Load1,27,289,50,14
END


/////////////////////////////////////////////////////////////////////////////
//
// Version
//

VS_VERSION_INFO VERSIONINFO
 FILEVERSION 1,0,0,1
 PRODUCTVERSION 1,0,0,1
 FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
 FILEFLAGS 0x1L
#else
 FILEFLAGS 0x0L
#endif
```

```
 FILEOS 0x4L
 FILETYPE 0x1L
 FILESUBTYPE 0x0L
BEGIN
   BLOCK "StringFileInfo"
   BEGIN
      BLOCK "040904e4"
      BEGIN
         VALUE "FileDescription", "Pathfinding Algorithm Testbed"
         VALUE "FileVersion", "1.0.0.1"
         VALUE "InternalName", "pathfinder.exe"
         VALUE "LegalCopyright", "TODO: (c) <Company name>.  All rights
reserved."
         VALUE "OriginalFilename", "pathfinder.exe"
         VALUE "ProductName", "Pathfinder"
         VALUE "ProductVersion", "1.0.0.1"
      END
   END
   BLOCK "VarFileInfo"
   BEGIN
      VALUE "Translation", 0x409, 1252
   END
END


/////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO
BEGIN
   IDD_ABOUTBOX, DIALOG
   BEGIN
      LEFTMARGIN, 7
      RIGHTMARGIN, 228
      TOPMARGIN, 7
      BOTTOMMARGIN, 48
   END

   IDD_TOOLBAR, DIALOG
   BEGIN
      RIGHTMARGIN, 120
      TOPMARGIN, 7
      BOTTOMMARGIN, 319
   END
END
#endif    // APSTUDIO_INVOKED
```

```
/////////////////////////////////////////////////////////////////////////////
//
// String Table
//

STRINGTABLE
BEGIN
    IDP_OLE_INIT_FAILED     "OLE initialization failed.  Make sure that the OLE
libraries are the correct version."
END

STRINGTABLE
BEGIN
    IDR_MAINFRAME
"pathfinder\n\npathfinder\n\n\npathfinder.Document\npathfinder.Document"
END

STRINGTABLE
BEGIN
    AFX_IDS_APP_TITLE       "pathfinder"
    AFX_IDS_IDLEMESSAGE     "Ready"
END

STRINGTABLE
BEGIN
    ID_INDICATOR_EXT        "EXT"
    ID_INDICATOR_CAPS       "CAP"
    ID_INDICATOR_NUM        "NUM"
    ID_INDICATOR_SCRL       "SCRL"
    ID_INDICATOR_OVR        "OVR"
    ID_INDICATOR_REC        "REC"
END

STRINGTABLE
BEGIN
    ID_FILE_NEW             "Create a new document\nNew"
    ID_FILE_OPEN            "Open an existing document\nOpen"
    ID_FILE_CLOSE           "Close the active document\nClose"
    ID_FILE_SAVE            "Save the active document\nSave"
    ID_FILE_SAVE_AS         "Save the active document with a new name\nSave As"
    ID_FILE_PAGE_SETUP      "Change the printing options\nPage Setup"
    ID_FILE_PRINT_SETUP     "Change the printer and printing options\nPrint Setup"
    ID_FILE_PRINT           "Print the active document\nPrint"
    ID_FILE_PRINT_PREVIEW   "Display full pages\nPrint Preview"
END

STRINGTABLE
BEGIN
```

```
   ID_APP_ABOUT          "Display program information, version number and
copyright\nAbout"
   ID_APP_EXIT         "Quit the application; prompts to save documents\nExit"
END

STRINGTABLE
BEGIN
   ID_FILE_MRU_FILE1       "Open this document"
   ID_FILE_MRU_FILE2       "Open this document"
   ID_FILE_MRU_FILE3       "Open this document"
   ID_FILE_MRU_FILE4       "Open this document"
   ID_FILE_MRU_FILE5       "Open this document"
   ID_FILE_MRU_FILE6       "Open this document"
   ID_FILE_MRU_FILE7       "Open this document"
   ID_FILE_MRU_FILE8       "Open this document"
   ID_FILE_MRU_FILE9       "Open this document"
   ID_FILE_MRU_FILE10      "Open this document"
   ID_FILE_MRU_FILE11      "Open this document"
   ID_FILE_MRU_FILE12      "Open this document"
   ID_FILE_MRU_FILE13      "Open this document"
   ID_FILE_MRU_FILE14      "Open this document"
   ID_FILE_MRU_FILE15      "Open this document"
   ID_FILE_MRU_FILE16      "Open this document"
END

STRINGTABLE
BEGIN
   ID_NEXT_PANE          "Switch to the next window pane\nNext Pane"
   ID_PREV_PANE          "Switch back to the previous window pane\nPrevious Pane"
END

STRINGTABLE
BEGIN
   ID_WINDOW_SPLIT        "Split the active window into panes\nSplit"
END

STRINGTABLE
BEGIN
   ID_EDIT_CLEAR         "Erase the selection\nErase"
   ID_EDIT_CLEAR_ALL       "Erase everything\nErase All"
   ID_EDIT_COPY          "Copy the selection and put it on the Clipboard\nCopy"
   ID_EDIT_CUT          "Cut the selection and put it on the Clipboard\nCut"
   ID_EDIT_FIND         "Find the specified text\nFind"
   ID_EDIT_PASTE         "Insert Clipboard contents\nPaste"
   ID_EDIT_REPEAT         "Repeat the last action\nRepeat"
   ID_EDIT_REPLACE        "Replace specific text with different text\nReplace"
   ID_EDIT_SELECT_ALL      "Select the entire document\nSelect All"
   ID_EDIT_UNDO          "Undo the last action\nUndo"
   ID_EDIT_REDO          "Redo the previously undone action\nRedo"
END
```

```
STRINGTABLE
BEGIN
   ID_VIEW_STATUS_BAR      "Show or hide the status bar\nToggle StatusBar"
END

STRINGTABLE
BEGIN
   AFX_IDS_SCSIZE          "Change the window size"
   AFX_IDS_SCMOVE          "Change the window position"
   AFX_IDS_SCMINIMIZE      "Reduce the window to an icon"
   AFX_IDS_SCMAXIMIZE      "Enlarge the window to full size"
   AFX_IDS_SCNEXTWINDOW    "Switch to the next document window"
   AFX_IDS_SCPREVWINDOW    "Switch to the previous document window"
   AFX_IDS_SCCLOSE         "Close the active window and prompts to save the
documents"
END

STRINGTABLE
BEGIN
   AFX_IDS_SCRESTORE       "Restore the window to normal size"
   AFX_IDS_SCTASKLIST      "Activate Task List"
END

STRINGTABLE
BEGIN
   AFX_IDS_PREVIEW_CLOSE   "Close print preview mode\nCancel Preview"
END

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////


/////////////////////////////////////////////////////////////////////////////
// English (U.K.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENG)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_UK
#pragma code_page(1252)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE
BEGIN
```

```
    "resource.h\0"
END

2 TEXTINCLUDE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE
BEGIN
    "#define _AFX_NO_SPLITTER_RESOURCES\r\n"
    "#define _AFX_NO_OLE_RESOURCES\r\n"
    "#define _AFX_NO_TRACKER_RESOURCES\r\n"
    "#define _AFX_NO_PROPERTY_RESOURCES\r\n"
    "\r\n"
    "#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)\r\n"
    "LANGUAGE 9, 1\r\n"
    "#pragma code_page(1252)\r\n"
    "#include ""res\\pathfinder.rc2""  // non-Microsoft Visual C++ edited resources\r\n"
    "#include ""afxres.rc""         // Standard components\r\n"
    "#include ""afxprint.rc""       // printing/print preview resources\r\n"
    "#endif\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.

#endif    // English (U.K.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
#define _AFX_NO_SPLITTER_RESOURCES
#define _AFX_NO_OLE_RESOURCES
#define _AFX_NO_TRACKER_RESOURCES
```

```
#define _AFX_NO_PROPERTY_RESOURCES

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
LANGUAGE 9, 1
#pragma code_page(1252)
#include "res\pathfinder.rc2"  // non-Microsoft Visual C++ edited resources
#include "afxres.rc"         // Standard components
#include "afxprint.rc"       // printing/print preview resources
#endif

/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

pathfinderDoc.h

```
///////////////////////////////////////////////////////////////////
// pathfinderDoc.h : interface of the CpathfinderDoc class
///////////////////////////////////////////////////////////////////


#pragma once


class CpathfinderDoc : public CDocument
{
protected: // create from serialization only
	CpathfinderDoc();
	DECLARE_DYNCREATE(CpathfinderDoc)

// Attributes
public:

// Operations
public:

// Overrides
public:
	virtual BOOL OnNewDocument();
	virtual void Serialize(CArchive& ar);

// Implementation
public:
	virtual ~CpathfinderDoc();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
	DECLARE_MESSAGE_MAP()
};
```

pathfinderDoc.cpp

```
/////////////////////////////////////////////////////////////////////////////
// pathfinderDoc.cpp : implementation of the CpathfinderDoc class
/////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "pathfinder.h"

#include "pathfinderDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CpathfinderDoc

IMPLEMENT_DYNCREATE(CpathfinderDoc, CDocument)

BEGIN_MESSAGE_MAP(CpathfinderDoc, CDocument)
END_MESSAGE_MAP()


// CpathfinderDoc construction/destruction

CpathfinderDoc::CpathfinderDoc()
{
	// TODO: add one-time construction code here

}

CpathfinderDoc::~CpathfinderDoc()
{
}

BOOL CpathfinderDoc::OnNewDocument()
{
	if (!CDocument::OnNewDocument())
		return FALSE;

	// TODO: add reinitialization code here
	// (SDI documents will reuse this document)

	return TRUE;
}
```

```
// CpathfinderDoc serialization

void CpathfinderDoc::Serialize(CArchive& ar)
{
	if (ar.IsStoring())
	{
		// TODO: add storing code here
	}
	else
	{
		// TODO: add loading code here
	}
}


// CpathfinderDoc diagnostics

#ifdef _DEBUG
void CpathfinderDoc::AssertValid() const
{
	CDocument::AssertValid();
}

void CpathfinderDoc::Dump(CDumpContext& dc) const
{
	CDocument::Dump(dc);
}
#endif //_DEBUG


// CpathfinderDoc commands
```

pathfinderView.h

```
/////////////////////////////////////////////////////////////////////////////
// pathfinderView.h : interface of the CpathfinderView class
/////////////////////////////////////////////////////////////////////////////

#pragma once

class CpathfinderView : public CView
{
protected: // create from serialization only
	CpathfinderView();
	DECLARE_DYNCREATE(CpathfinderView)

// Attributes
public:
	CpathfinderDoc* GetDocument() const;

// Operations
public:

// Overrides
public:
	virtual void OnDraw(CDC* pDC);  // overridden to draw this view
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

// Implementation
public:
	virtual ~CpathfinderView();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
	afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
	DECLARE_MESSAGE_MAP()

};

#ifndef _DEBUG  // debug version in pathfinderView.cpp
inline CpathfinderDoc* CpathfinderView::GetDocument() const
   { return reinterpret_cast<CpathfinderDoc*>(m_pDocument); }
#endif
```

pathfinderView.cpp

```
////////////////////////////////////////////////////////////////////////////
// pathfinderView.cpp : implementation of the CpathfinderView class
////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "pathfinder.h"

#include "pathfinderDoc.h"
#include "pathfinderView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CpathfinderView

IMPLEMENT_DYNCREATE(CpathfinderView, CView)

BEGIN_MESSAGE_MAP(CpathfinderView, CView)
        ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()

// CpathfinderView construction/destruction

CpathfinderView::CpathfinderView()
{
        // TODO: add construction code here

}

CpathfinderView::~CpathfinderView()
{
}

BOOL CpathfinderView::PreCreateWindow(CREATESTRUCT& cs)
{
        // TODO: Modify the Window class or styles here by modifying
        //  the CREATESTRUCT cs

        return CView::PreCreateWindow(cs);
}

// CpathfinderView drawing

void CpathfinderView::OnDraw(CDC* pDC)
{
        CpathfinderDoc* pDoc = GetDocument();
```

```
	ASSERT_VALID(pDoc);
	if (!pDoc)
		return;

	//TODO: add draw code for native data here
}

void CpathfinderView::OnLButtonDown(UINT nFlags,CPoint point){
   MessageBox(_T("LEFT MOUSE BUTTON PRESSED!!!"));
}

// CpathfinderView diagnostics

#ifdef _DEBUG
void CpathfinderView::AssertValid() const
{
	CView::AssertValid();
}

void CpathfinderView::Dump(CDumpContext& dc) const
{
	CView::Dump(dc);
}

CpathfinderDoc* CpathfinderView::GetDocument() const // non-debug version is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CpathfinderDoc)));
	return (CpathfinderDoc*)m_pDocument;
}
#endif //_DEBUG
```

mainFrm.h

```
/////////////////////////////////////////////////
// MainFrm.h : interface of the CMainFrame class
/////////////////////////////////////////////////

#include <afxwin.h>
#include "node.h"

//Various states of the application
#define STATUS_PAUSED 0;
#define STATUS_PLACE_START 1;
#define STATUS_PLACE_END 2;
#define STATUS_PLACE_WALL 3;
#define STATUS_PLACE_SPACE 4;
#define STATUS_GO 5;

#define MAX_LENGTH 15;
#define MAX_WIDTH 15;

#pragma once

//Class main frame inherits from CFrameWnd
class CMainFrame : public CFrameWnd
{

public: // create from serialization only
	CMainFrame();
	DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:
	int iSTATUS;


// Operations
public:
	node node_graph[21][21], null_node;

	//pointers to the start and end nodes
	node *start_node, *end_node, *change_node;

private:
	//used for threading
	char* TempChar;

// Overrides
public:
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
```

```
          void updateStatus();
          void updateGrid();
          void initNodeGraph();
          void resetGraph();
          void simpleResetGraph();
          void setStatus(int status);
          int getStatus();
          bool lineofsight(int x, int y);
          CPoint moveToPoint(CPoint current, CPoint dest);
          int setDirection(CPoint current, CPoint dest);
          CPoint moveDirection(CPoint current,int direction);
          bool pointInGraph(CPoint point);
          int drawLine();
          void drawLine2();
          float calcHeuristic(CPoint source, CPoint destination);
          void load_maze(int maze_num);
          bool pathGenerated();

// Implementation
public:
          virtual ~CMainFrame();
#ifdef _DEBUG
          virtual void AssertValid() const;
          virtual void Dump(CDumpContext& dc) const;
#endif

protected:  // control bar embedded members
          CStatusBar  m_wndStatusBar;
          CDialogBar  m_wndToolBar;
          CComboBox *pComboAlgorithm;
          CMenu *pMainMenu;
          CEdit *ptestbox;
          CEdit *ptestbox2;
          CEdit *pstatusbox;

// Generated message map functions
protected:
          afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
          afx_msg void OnMouseMove(UINT nFlags, CPoint point);
          afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
          afx_msg void OnPaint();

          DECLARE_MESSAGE_MAP()

public:
          afx_msg void OnBnClickedBtnGo();
          afx_msg void OnCbnSelchangeCombo1();
          afx_msg void OnBnClickedBtnPause();
          afx_msg void OnBnClickedBtnReset();
          afx_msg void OnBnClickedBtnStart();
```

```
	afx_msg void OnBnClickedBtnEnd();
	afx_msg void OnBnClickedBtnWall();
	afx_msg void OnBnClickedBtnClear();
public:
	afx_msg void OnBnClickedBtnCleargraph();
public:
	afx_msg void OnBnClickedLoad1();
};
```

mainfrm.cpp

```
///////////////////////////////////////////////////////
// MainFrm.cpp : implementation of the CMainFrame class
///////////////////////////////////////////////////////

#include "windows.h"
#include "stdafx.h"
#include "math.h"
#include "pathfinder.h"
#include "afxwin.h"
#include "node.h"
#include "priorityQueue.h"
#include "performanceTracker.h"
#include <cstdlib>
#include <vector>
#include <queue>
#include <deque>
#include <list>
#include <iostream>

#include "MainFrm.h"

//define status
#define STATUS_PAUSED 0;
#define STATUS_PLACE_START 1;
#define STATUS_PLACE_END 2;
#define STATUS_PLACE_WALL 3;
#define STATUS_PLACE_SPACE 4;
#define STATUS_GO 5;
#define STATUS_CLEAR 6;
#define STATUS_RESET 7;

//define node contents
#define NODE_SPACE 0;
#define NODE_WALL 1;
#define NODE_START 2;
#define NODE_END 3;

#ifdef _DEBUG
#define new DEBUG_NEW
#endif
// CMainFrame

//Algorithm Test!!!
UINT randombounce2( LPVOID pParam );
UINT wallTrace(LPVOID pParam);
UINT breadthFirstSearch(LPVOID pParam);
UINT dijkstraSearch(LPVOID pParam);
UINT bestFirstSearch(LPVOID pParam);
UINT aStar(LPVOID pParam);
UINT dynamicAstar(LPVOID pParam);
UINT environmentSubtraction(LPVOID pParam);

//My Coloured Brushes
CBrush brushRed(RGB(255, 2, 5));
CBrush brushBlack(RGB(0, 0, 0));
CBrush brushGreen(RGB(0, 125, 5));
CBrush brushWhite(RGB(255,255,255));

//My Coloured Pens
CPen BlueThickPen(PS_SOLID, 2, RGB(0, 0, 255));
CPen BlackThickPen(PS_SOLID, 2, RGB(0, 0, 0));
CPen BlackThinPen(PS_SOLID, 1, RGB(0, 0, 0));
CPen GreenThickPen(PS_SOLID, 2, RGB(0, 125, 5));
CPen RedThickPen(PS_SOLID, 2, RGB(255, 0, 0));

//vector used to store the generated path
std::list <node *> generatedPath;

//Maze data
int maze1[21][21] = {
                {0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,1},
                {0,1,1,0,1,1,1,0,1,0,1,1,1,0,1,1,0,1,0,0,1},
                {0,1,1,0,1,1,1,0,1,0,1,1,1,0,1,1,0,1,0,0,1},
                {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1},
                {0,1,1,0,1,0,1,1,1,1,1,0,1,0,1,1,0,1,0,0,1},
                {0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,1,0,0,1},
                {1,1,1,0,1,1,1,0,1,0,1,1,1,0,1,1,1,1,0,0,1},
                {0,0,1,0,1,0,0,0,0,0,0,0,1,0,1,0,0,1,0,0,1},
                {1,1,1,0,1,0,1,1,0,1,1,0,1,0,1,1,1,1,0,0,1},
                {0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,1},
                {1,1,1,0,1,0,1,1,0,1,1,0,1,0,1,1,1,1,0,0,1},
                {0,0,1,0,1,0,0,0,0,0,0,0,1,0,1,0,0,1,0,0,1},
                {1,1,1,0,1,0,1,1,1,1,1,0,1,0,1,1,1,1,0,0,1},
                {0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,1},
                {0,1,1,0,1,1,1,0,1,0,1,1,1,0,1,1,0,1,0,0,1},
                {0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,1},
                {1,0,1,0,1,0,1,1,1,1,1,0,1,0,1,0,1,1,0,0,1},
                {0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,1,0,0,1},
                {0,1,1,1,1,1,1,0,1,0,1,1,1,1,1,1,0,1,0,0,1},
                {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1},
                {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}};

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

//Message maps
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
        ON_WM_PAINT()
        ON_WM_CREATE()
        ON_WM_LBUTTONDOWN()
        ON_BN_CLICKED(IDC_BTN_GO, &CMainFrame::OnBnClickedBtnGo)
        ON_BN_CLICKED(IDC_BTN_PAUSE,
&CMainFrame::OnBnClickedBtnPause)
        ON_BN_CLICKED(IDC_BTN_RESET,
&CMainFrame::OnBnClickedBtnReset)
        ON_BN_CLICKED(IDC_BTN_START,
&CMainFrame::OnBnClickedBtnStart)
        ON_BN_CLICKED(IDC_BTN_END, &CMainFrame::OnBnClickedBtnEnd)
        ON_BN_CLICKED(IDC_BTN_WALL,
&CMainFrame::OnBnClickedBtnWall)
        ON_BN_CLICKED(IDC_BTN_CLEAR,
&CMainFrame::OnBnClickedBtnClear)
        ON_BN_CLICKED(IDC_BTN_CLEARGRAPH,
&CMainFrame::OnBnClickedBtnCleargraph)
        ON_BN_CLICKED(IDC_Load1, &CMainFrame::OnBnClickedLoad1)
END_MESSAGE_MAP()

static UINT indicators[] =
{
        ID_SEPARATOR,           // status line indicator
        ID_INDICATOR_CAPS,
        ID_INDICATOR_NUM,
        ID_INDICATOR_SCRL,
};

////////////////////////////////////////
// CMainFrame construction/destruction
////////////////////////////////////////
CMainFrame::CMainFrame()
{
  // Create the window's frame
  Create(NULL, _T("Pathfinder ver 1.2 by Daniel Potter"),
        WS_OVERLAPPED | WS_CAPTION  | WS_SYSMENU |
WS_THICKFRAME,
         CRect(100, 100, 900, 780), NULL);

  //set the status
  iSTATUS = 0;
  updateStatus();

  //set up the null node
  null_node.setPosition(1000,1000);

  //initialise the node graph
  initNodeGraph();
}

CMainFrame::~CMainFrame()
```

```
{

}

//This method is run when the create message is recieved
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{

        if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
                return -1;

   //Create the status bar
        if (!m_wndStatusBar.Create(this) ||
                !m_wndStatusBar.SetIndicators(indicators,
                  sizeof(indicators)/sizeof(UINT)))
        {
                Sleep(2000);
                TRACE0("Failed to create status bar");
                return -1;      // fail to create
        }

        //Create the toolbar
        if( !m_wndToolBar.Create(this, IDD_TOOLBAR, CBRS_RIGHT |
CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY, IDD_TOOLBAR) )
        {

                TRACE0("Failed to create the Tool bar");
                return -1;      // fail to create
        }

   this->m_wndToolBar.SetWindowText(TEXT("Tool Bar"));

        m_wndToolBar.EnableDocking(CBRS_ALIGN_LEFT);
        //DockControlBar(&m_wndToolBar);


        //Pointer to the combobox
        pComboAlgorithm = reinterpret_cast<CComboBox *>(this-
>m_wndToolBar.GetDlgItem(IDC_COMBO1));
        pComboAlgorithm->SetWindowText(_T("Please Select Algorithm"));
        pComboAlgorithm->AddString(_T("Random Bounce"));
        pComboAlgorithm->AddString(_T("Wall Trace"));
        pComboAlgorithm->AddString(_T("Breadth First Search"));
        pComboAlgorithm->AddString(_T("DijkStra"));
        pComboAlgorithm->AddString(_T("Best First Search"));
        pComboAlgorithm->AddString(_T("A* (A-Star)"));
        pComboAlgorithm->AddString(_T("D* (Dynamic A-Star)"));
        pComboAlgorithm->AddString(_T("Environment Subtraction"));
   ptestbox = reinterpret_cast<CEdit *>(this-
>m_wndToolBar.GetDlgItem(IDC_TESTBOX));
        ptestbox->SetWindowText(_T("12000"));

   updateStatus();


        return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
        if( !CFrameWnd::PreCreateWindow(cs) )
                return FALSE;
        // TODO: Modify the Window class or styles here by modifying
        //  the CREATESTRUCT cs


        //set the size of the window
        //cs.cx = 945;
        //cs.cy = 807;

        return TRUE;
}


// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
        CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
        CFrameWnd::Dump(dc);
}

#endif //_DEBUG


// CMainFrame message handlers



////////////////////////////////////////
//When the go button is pressed
////////////////////////////////////////
void CMainFrame::OnBnClickedBtnGo()
```

```
{
        bool clear = false;

        // TODO: Add your control notification handler code here
        int iIndexNo;

        //get a pointer to the testbox
        ptestbox = reinterpret_cast<CEdit *>(this-
>m_wndToolBar.GetDlgItem(IDC_TESTBOX));
        ptestbox2 = reinterpret_cast<CEdit *>(this-
>m_wndToolBar.GetDlgItem(IDC_TESTBOX2));
        pComboAlgorithm = (CComboBox *) this-
>m_wndToolBar.GetDlgItem(IDC_COMBO1);

        //get the index number of the item selected from the combo box
        iIndexNo = pComboAlgorithm->GetCurSel();

        if(start_node->getPosition() == null_node.getPosition()){
          MessageBox(_T("Start Node Not placed, Please Place one"));
                clear = false;
        }
        else if(end_node->getPosition() == null_node.getPosition()){
          MessageBox(_T("End Node Not placed, Please Place one"));
                clear = false;
        }
        else if(iIndexNo < 0){
          MessageBox(_T("No Pathfinding Algorithm Selected, Please choose one"));
                clear = false;
        }
        else{
          clear = true;
        }

  //if the index number is less than 0 then it is not a valid selection
        if(!(iIndexNo < 0)&&(clear)){

                iSTATUS = STATUS_GO;
                updateStatus();

    if( iIndexNo == CB_ERR )
                {

                }
                //random bounce selected
                else if(iIndexNo == 0){
                        AfxBeginThread(randombounce2,this);
                }
    //wall trace selected
                else if(iIndexNo == 1){
                        AfxBeginThread(wallTrace,this);
                }
                //Best First Search selected
                else if(iIndexNo == 2){
                        AfxBeginThread(breadthFirstSearch,this);
                }
        //Dijkstra Search
                else if(iIndexNo == 3){
                        AfxBeginThread(dijkstraSearch,this);
                }
                //Best First Search
                else if(iIndexNo == 4){
                        AfxBeginThread(bestFirstSearch,this);
                }
                //Best First Search
                else if(iIndexNo == 5){
                        AfxBeginThread(aStar,this);
                }
                else if(iIndexNo == 6){
                        AfxBeginThread(dynamicAstar,this);
                }
                else if(iIndexNo == 7){
                        MessageBox(_T("In order to use this algorithm you must
\n1.Generate a path using another pathfinding algorithm \n2.Select this algorithm
\n3.Clear a space on the graph "));
                }
        }

        UpdateData(FALSE);
        this->RedrawWindow();
}

/////////////////////////////////
////PAUSE THE PATHFINDING PROCESS
void CMainFrame::OnBnClickedBtnPause()
{
        iSTATUS = STATUS_PAUSED;
        updateStatus();
}

////////////////////////////////////////
//Update the status box
////////////////////////////////////////
void CMainFrame::updateStatus(){

    CString test;

    pstatusbox = reinterpret_cast<CEdit *>(this-
>m_wndToolBar.GetDlgItem(IDC_STATUS));
    ptestbox = reinterpret_cast<CEdit *>(this-
>m_wndToolBar.GetDlgItem(IDC_TESTBOX));
```

```
    test.Format(_T("Status = %d"),iSTATUS);
        ptestbox->SetWindowText(test);

        if(iSTATUS == 0){
                        pstatusbox->SetWindowTextW(_T("PAUSED"));
        }
        else if(iSTATUS == 1){
                        pstatusbox->SetWindowTextW(_T("PLACE START"));
        }
        else if(iSTATUS == 2){
                        pstatusbox->SetWindowTextW(_T("PLACE END"));
        }
    else if(iSTATUS == 3){
                        pstatusbox->SetWindowTextW(_T("PLACE WALL"));
        }
        else if(iSTATUS == 4){
                        pstatusbox->SetWindowTextW(_T("PLACE SPACE"));
        }
        else if(iSTATUS == 5){
                        pstatusbox->SetWindowTextW(_T("PERFORMING"));
        }
        else if(iSTATUS == 7){
                        pstatusbox->SetWindowTextW(_T("CLEARING"));
        }

}

///////////////////////////////
//Initialise the node graph
///////////////////////////////
void CMainFrame::initNodeGraph(){

        int x,y;

    for(x = 0; x <= 20; x++){
                for(y = 0; y <= 20; y++){

                        if((y==20)||(x==20))
                                node_graph[x][y].set_contents(1);
                        else
                                node_graph[x][y].clearNode();

                        node_graph[x][y].setPosition(x,y);
                        node_graph[x][y].setDistance(0.0);
                        node_graph[x][y].setHeuristic(0.0);
                        node_graph[x][y].setParentNode(x,y);
                        node_graph[x][y].setSearched(false);
                        node_graph[x][y].set_state(0);
                        node_graph[x][y].onShortestPath(false);
```

```
            }
        }

        start_node = &null_node;
        end_node = &null_node;
}

////////////////////////////////
//Reset the graph
////////////////////////////////

void CMainFrame::resetGraph(){
   int x,y;

   for(x = 0; x <= 20; x++){
                for(y = 0; y <= 20; y++){

                        node_graph[x][y].setPosition(x,y);
                        node_graph[x][y].setDistance(0.0);
                        node_graph[x][y].setHeuristic(0.0);
                        node_graph[x][y].setParentNode(x,y);
                        node_graph[x][y].setSearched(false);
                        node_graph[x][y].onShortestPath(false);
                        node_graph[x][y].set_state(0);
                }
        }
}

///////////////////////////////////////////////////////////////////////////
//Reset some of the aspects of the nodes
//This is used by the dynamic pathfinding algorithms
///////////////////////////////////////////////////////////////////////////

void CMainFrame::simpleResetGraph(){
   int x,y;

   for(x = 0; x <= 20; x++){
                for(y = 0; y <= 20; y++){
                        //node_graph[x][y].setDistance(0.0);
                        node_graph[x][y].setHeuristic(0.0);
                        node_graph[x][y].setSearched(false);
                }
        }
}

////////////////////////////////
//Is there a generated path?
////////////////////////////////
bool CMainFrame::pathGenerated(){
        if(generatedPath.empty())
```

```
            return false;
        else
                return true;
}

////////////////////////////////
//Function to paint the window//
////////////////////////////////
void CMainFrame::OnPaint(){

        //paint device context
   CPaintDC dc(this);

        updateGrid();
}

/////////////////////////////////////////////
//Updates the grid
/////////////////////////////////////////////
void CMainFrame::updateGrid(){

        CClientDC dc(this);
        CRect circle;

   int x,y;

        //go through each node in the graph
        for(x = 0; x< 20; x++){
                for(y = 0; y< 20; y++){
                        dc.SelectObject(&BlackThinPen);
                        //if the node is a wall
                        if(node_graph[x][y].get_contents() == 1){
            dc.SelectObject(&brushBlack);
                                dc.Rectangle(x*30,y*30,x*30+30,y*30+30);
                        }
                        //if the node is the start
                        else if(node_graph[x][y].get_contents() == 2){
                                dc.SelectObject(&brushWhite);
                                dc.Rectangle(x*30,y*30,x*30+30,y*30+30);
                                dc.SelectObject(&BlackThickPen);
                                dc.SelectObject(&brushGreen);
                                circle.SetRect(x*30,y*30,x*30+30,y*30+30);
                                dc.Ellipse(&circle);
                        }
                        //if the node is the end
                        else if(node_graph[x][y].get_contents() == 3){
                                dc.SelectObject(&brushWhite);
                                dc.Rectangle(x*30,y*30,x*30+30,y*30+30);
                                dc.SelectObject(&BlackThickPen);
                                dc.SelectObject(&brushRed);
                        circle.SetRect(x*30,y*30,x*30+30,y*30+30);
                        dc.Ellipse(&circle);
                    }
                    //if the node is a space
                    else if(node_graph[x][y].get_contents() == 0){
                        dc.SelectObject(&brushWhite);
                        dc.Rectangle(x*30,y*30,x*30+30,y*30+30);
                    }
            }
        }

        drawLine2();
}


/////////////////////////////////////////
//RESET BUTTON PRESSED
////////////////////////////////////////
void CMainFrame::OnBnClickedBtnReset()
{
        iSTATUS = STATUS_RESET;
        resetGraph();
        updateGrid();
        //clear the generated path
        generatedPath.clear();
}

//////////////////////////////////////
//get the current status of the application
//////////////////////////////////////
int CMainFrame::getStatus(){
  return iSTATUS;
}

///////////////////////////////////////
//Set the status of the applcation
///////////////////////////////////////
void CMainFrame::setStatus(int status){
  iSTATUS = status;
}

//////////////////////////////////
//LEFT MOUSE BUTTON PRESSED
/////////////////////////////////
void CMainFrame::OnLButtonDown(UINT nFlags,CPoint point){

  int x,y;

        //Convert the input position
        x = point.x / 30;
        y = point.y / 30;

        //perform the input on the graph
    if(iSTATUS == 0){
        }
        //add the start node
        else if(iSTATUS == 1){
                    //set selected nodes contents to the start
                        node_graph[x][y].set_contents(2);
                        //set previous start contents back to a space
                        start_node->set_contents(0);
                        //set the pointer to the start node
                        start_node = &node_graph[x][y];
        }
        //add the end node
        else if(iSTATUS == 2){
                        node_graph[x][y].set_contents(3);
                        end_node->set_contents(0);
                        end_node = &node_graph[x][y];
        }
        //add a wall
    else if(iSTATUS == 3){
                        node_graph[x][y].set_contents(1);
        }
        //clear a space, if the space cleared was a wall and the algorithm currently
selected is the subtraction dynamic
        //pathfinding algorithm
        else if(iSTATUS == 4){
                //get a pointer to the combo box
                pComboAlgorithm = (CComboBox *) this-
>m_wndToolBar.GetDlgItem(IDC_COMBO1);

                //if the algorithm subtraction change has been selected from the
combobox and a change has been made
                if(pComboAlgorithm->GetCurSel()==7){

        //if the contents of the node that we are clearing is a wall then we need to
recalculate the path
                        if(node_graph[x][y].get_contents()==1){
                                node_graph[x][y].set_contents(0);
                                //set the node at where the change in the graph was made
                                change_node = &node_graph[x][y];

                                updateGrid();

                                //begin processing the algorithm
                                AfxBeginThread(environmentSubtraction,this);

                        }
                        else{
```

```
                        node_graph[x][y].set_contents(0);
                    }
                }else{
                    node_graph[x][y].set_contents(0);
                };


        }
        else if(iSTATUS == 5){
        }

        //updateGrid();

        PostMessage(WM_PAINT,NULL,NULL);
}


//////////////////////////////////////
//When the start node button is placed
//////////////////////////////////////
void CMainFrame::OnBnClickedBtnStart()
{
        if(iSTATUS != 5){
                iSTATUS = STATUS_PLACE_START;
                updateStatus();
        };
}

//////////////////////////////////////
//When the end button is pressed
//////////////////////////////////////
void CMainFrame::OnBnClickedBtnEnd()
{
        if(iSTATUS != 5){
                iSTATUS = STATUS_PLACE_END;
                updateStatus();
        };
}

////////////////////////////////////////
//When the Wall Button is pressed
///////////////////////////////////////
void CMainFrame::OnBnClickedBtnWall()
{
        if(iSTATUS != 5){
                iSTATUS = STATUS_PLACE_WALL;
                updateStatus();
        };
}
```

```
///////////////////////////////////////
//When the clear button is pressed
///////////////////////////////////////
void CMainFrame::OnBnClickedBtnClear()
{
        if(iSTATUS != 5){
                iSTATUS = STATUS_PLACE_SPACE;
                updateStatus();
        };
}


///////////////////////////////////////////////////
//Clear the entire graph and set it all to spaces
///////////////////////////////////////////////////
void CMainFrame::OnBnClickedBtnCleargraph()
{
        iSTATUS = STATUS_CLEAR;
        initNodeGraph();
        PostMessage(WM_PAINT,NULL,NULL);
        updateStatus();
        //clear the generated path
        generatedPath.clear();
}

////////////////////////////////////////////////////
//What happens when the load button is pressed
////////////////////////////////////////////////////
void CMainFrame::OnBnClickedLoad1()
{
   //load the maze
        load_maze(1);

        //reset the start and end nodes
        start_node = &null_node;
        end_node = &null_node;
}

///////////////////////////////////////////////////////
//transefer data from precreated graphs to the main graph
///////////////////////////////////////////////////////
void CMainFrame::load_maze(int maze_num){

        for(int x = 0;x<=20;x++){
                for(int y = 0;y<=20;y++){
                        node_graph[y][x].set_contents(maze1[x][y]);
                }
        }

        updateGrid();
```

```
}

////////////////////////////////////////////
//Check To See if a line of sight exists
////////////////////////////////////////////
bool CMainFrame::lineofsight(int x, int y){

        CPoint tempPos,endPos;
        CString string;

        endPos = end_node->getPosition();

        do{

                //move the x and y co-ordinates towards the end position
                if(x > endPos.x){
                  x--;
                }
                else if(x < endPos.x){
                  x++;
                }

                if(y > endPos.y){
                  y--;
                }
                else if(y < endPos.y){
                  y++;
                }

                //if at any point the node is the goal return true
                if(node_graph[x][y].get_contents() == 3){
                    return true;
                }
                //else if at any point the node is a wall return false
                else if(node_graph[x][y].get_contents() == 1){
                    return false;
                }

        }while((node_graph[x][y].get_contents() != 3)||(node_graph[x][y].get_contents()
!= 1));
   return 0;
}

/////////////////////////////////////////////////////////
//This Function returns the next point to move to in a move to point algorithm
/////////////////////////////////////////////////////////
CPoint CMainFrame::moveToPoint(CPoint current, CPoint dest){

        if((dest.x > current.x)&&(dest.y < current.y)){ //direction northeast
        current.x++;
            current.y--;
    }
    else if((dest.x > current.x)&&(dest.y > current.y)){ //direction Southeast
            current.x++;
            current.y++;
    }
    else if((dest.x < current.x)&&(dest.y > current.y)){ //direction southwest
            current.x--;
            current.y++;
    }
    else if((dest.x < current.x)&&(dest.y < current.y)){ //direction Northwest
            current.x--;
            current.y--;
    }
    else if(dest.y < current.y){ //direction north
            current.y--;
    }
    else if(dest.x > current.x){ //direction east
            current.x++;
    }
    else if(dest.y > current.y){ //direction south
  current.y++;
    }
    else if(dest.x < current.x){ //direction west
            current.x--;
    }

    return current;
}

/////////////////////////////////////////////////////////////////
//This Function returns the next point to move to in a move to point algorithm
/////////////////////////////////////////////////////////////////
int CMainFrame::setDirection(CPoint current, CPoint dest){

    if((dest.x > current.x)&&(dest.y < current.y)){ //direction northeast
        return 2;
    }
    else if((dest.x > current.x)&&(dest.y > current.y)){ //direction Southeast
            return 4;
    }
    else if((dest.x < current.x)&&(dest.y > current.y)){ //direction southwest
            return 6;
    }
    else if((dest.x < current.x)&&(dest.y < current.y)){ //direction Northwest
            return 8;
    }
    else if(dest.y < current.y){ //direction north
            return 1;
		}
		else if(dest.x > current.x){ //direction east
			return 3;
		}
		else if(dest.y > current.y){ //direction south
	return 5;
		}
		else if(dest.x < current.x){ //direction west
			return 7;
		}
}

////////////////////////////////////////////////////////////
//Move the point in the direction
////////////////////////////////////////////////////////////
CPoint CMainFrame::moveDirection(CPoint current, int direction){
	if(direction == 2){ //direction northeast
		   current.x++;
			current.y--;
		}
		else if(direction == 4){ //direction Southeast
			current.x++;
			current.y++;
		}
		else if(direction == 6){ //direction southwest
			current.x--;
			current.y++;
		}
		else if(direction == 8){ //direction Northwest
			current.x--;
			current.y--;
		}
		else if(direction == 1){ //direction north
			current.y--;
		}
		else if(direction == 3){ //direction east
			current.x++;
		}
		else if(direction == 5){ //direction south
	current.y++;
		}
		else if(direction == 7){ //direction west
			current.x--;
		}

		return current;
}

////////////////////////////////////////////////////////////
//Check to see if this point is on the graph
```

```
//////////////////////////////////////////////////
bool CMainFrame::pointInGraph(CPoint point){
        if((point.x>=0)&&(point.x<20)){
           if((point.y>=0)&&(point.y<20)){
              return true;
           }
        }
        return false;
}

//////////////////////////////////////////////////
//Draw a line from the end to the start and count the number of nodes and record all the
nodes on the generated path
//////////////////////////////////////////////////
int CMainFrame::drawLine(){

        CClientDC dc(this);
        CPoint startPos,endPos,tempPos,previousPos;

        int total = 0;

        startPos = start_node->getPosition();
        endPos = end_node->getPosition();

        tempPos = endPos;

        dc.SelectObject(&RedThickPen);

        //clear the generated path
        generatedPath.clear();

        do{
                generatedPath.push_front(&node_graph[tempPos.x][tempPos.y]);
                total++;
                previousPos = tempPos;
                tempPos = node_graph[tempPos.x][tempPos.y].getParentNode();
                node_graph[tempPos.x][tempPos.y].onShortestPath(true);
      dc.MoveTo(tempPos.x*30+15,tempPos.y*30+15);
           dc.LineTo(previousPos.x*30+15,previousPos.y*30+15);
        }while(node_graph[tempPos.x][tempPos.y].get_contents() != 2);

   return total;

}

//////////////////////////////////////////////////
//Draw a line from the end to the start
//////////////////////////////////////////////////
void CMainFrame::drawLine2(){
        if(!generatedPath.empty()){

                CClientDC dc(this);
                CPoint tempPos,previousPos;
                node *ptrNode;

                ptrNode = generatedPath.back();

                tempPos = ptrNode->getPosition();

                dc.SelectObject(&RedThickPen);

                while(node_graph[tempPos.x][tempPos.y].get_contents() != 2){
                        previousPos = tempPos;
                        tempPos = node_graph[tempPos.x][tempPos.y].getParentNode();
                        dc.MoveTo(tempPos.x*30+15,tempPos.y*30+15);
                        dc.LineTo(previousPos.x*30+15,previousPos.y*30+15);
                }
        }

}

/////////////////////////////////////////////////////////////
//Calculate the heuristic between to points
/////////////////////////////////////////////////////////////
float CMainFrame::calcHeuristic(CPoint source, CPoint destination){
   float temp;
        temp = (((destination.x - source.x)*(destination.x - source.x))+((destination.y -
source.y)*(destination.y - source.y)));
        //get the squareroot
        temp = sqrt(temp);
        return temp;
}

///////////////////////////////////////////////////////////
//The Random Bounce Algorithm
///////////////////////////////////////////////////////////
UINT randombounce2(LPVOID pParam){

        //get a pointer to the frame class that this being derived from!!
   CMainFrame* threadFrame = (CMainFrame*)pParam;

   CClientDC dc(threadFrame);
   CPoint currentPos, previousPos, tempPos;
        CBrush brushBlue(RGB(0, 0, 255));
   bool complete = false;
        int direction = 0;
        CString string;
        CPoint start_node,end_node;
```

```
threadFrame->updateGrid();

//get the position of the start node
currentPos = threadFrame->start_node->getPosition();
start_node = currentPos;
end_node = threadFrame->end_node->getPosition();

//used to track the performance of the algorithm
performanceTracker algTrack;

//perform the algorithm while it isnt complete or reset hasnt been pressed
do{
        //if the algorithm isnt paused
        if(threadFrame->iSTATUS != 0){

                algTrack.addSearch();
                algTrack.addStep();

                //generate a random number between 1 and 8
direction = 1+int(8*rand()/(RAND_MAX + 1.0));

tempPos = currentPos;

                if(direction == 1){ //direction north
  tempPos.y--;
                }
                else if(direction == 2){ //direction northeast
                   tempPos.y--; tempPos.x++;
                }
                else if(direction == 3){ //direction east
                   tempPos.x++;
                }
                else if(direction == 4){ //direction south east
                   tempPos.y++; tempPos.x++;
                }
                else if(direction == 5){ //direction South
                   tempPos.y++;
                }
                else if(direction == 6){ //direction South West
                   tempPos.y++; tempPos.x--;
                }
                else if(direction == 7){ //direction West
                   tempPos.x--;
                }
                else if(direction == 8){ //direction north west
                   tempPos.y--; tempPos.x--;
                }

                //check to see if not moving off the graph and that the node being
moved to isnt a wall
```

```
                        if(((tempPos.y >= 0)&&(tempPos.y <= 20)) && ((tempPos.x >=
0)&&(tempPos.x <= 20)) && (threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() != 1)){

                                        previousPos = currentPos;
                        currentPos = tempPos;

                                                threadFrame-
>node_graph[tempPos.x][tempPos.y].setParentNode(previousPos.x,previousPos.y);

                                                dc.SelectObject(&BlueThickPen);

                                                //draw a line from the previous node to the
destination node

dc.MoveTo(previousPos.x*30+15,previousPos.y*30+15);
                                    dc.LineTo(currentPos.x*30+15,currentPos.y*30+15);

                                                //If the current position is the start or end node
then we want to redraw the circle over it
                                                if((currentPos == start_node)||(previousPos ==
start_node)){
                                                   dc.SelectObject(&BlackThickPen);
                                                        dc.SelectObject(&brushGreen);

                                                        if(previousPos == start_node){

dc.Ellipse(previousPos.x*30,previousPos.y*30,previousPos.x*30+30,previousPos.y*30
+30);
                                                        }
                                                        else{

dc.Ellipse(currentPos.x*30,currentPos.y*30,currentPos.x*30+30,currentPos.y*30+30);
                                                        }
                                                }
                                                else if(currentPos == end_node){
                                                   dc.SelectObject(&BlackThickPen);
                                                        dc.SelectObject(&brushGreen);

dc.Ellipse(currentPos.x*30,currentPos.y*30,currentPos.x*30+30,currentPos.y*30+30);
                                                }
                              }

                              //if the current node in the search is the end node
                              if(threadFrame-
>node_graph[currentPos.x][currentPos.y].get_contents() == 3){
                          complete = true;
                                    algTrack.display(0);
                                    threadFrame->iSTATUS = 0;
                                    return 0;
```

```
                    }
                Sleep(200);
                  }
          }while((!complete)&&(threadFrame->iSTATUS != 7)&&(threadFrame-
>iSTATUS != 6));


   return 0;
}

//////////////////////////////
//Wall tracing algorithm
//////////////////////////////
UINT wallTrace(LPVOID pParam){

          //get a pointer to the frame class that this being derived from!!
    CMainFrame* threadFrame = (CMainFrame*)pParam;

    CClientDC dc(threadFrame);
    CPoint currentPos, previousPos, tempPos;
          CBrush brushBlue(RGB(0, 0, 255));
    bool complete = false,sight;
          int direction = 0,originalDirection = 0;
          float slope;
          CString string;
          CPoint start_node,end_node;

          //redraw the grid
          threadFrame->updateGrid();

          //Initialise all positions
          currentPos = threadFrame->start_node->getPosition();
          start_node = currentPos;
          end_node = threadFrame->end_node->getPosition();
          tempPos = currentPos;

          //slope = ((end_node.y-start_node.y)/(end_node.x-start_node.x));

          //used to track the performance of the algorithm
          performanceTracker algTrack;

          //perform the algorithm while it isnt complete or reset hasnt been pressed
          do{

            //move towards the goal node
            sight = false;
                 direction = threadFrame->setDirection(currentPos,end_node);
                 tempPos = threadFrame->moveDirection(currentPos,direction);

                 algTrack.addSearch();
```

```
                    //if the next space is a wall node we have to try and traverse around it
                    if((threadFrame->node_graph[tempPos.x][tempPos.y].get_contents() ==
1)||(!threadFrame->pointInGraph(tempPos))){
                              //get the original direction
          originalDirection = direction;

                              //repeat until obstacle has been traverse
                              do{
                                        algTrack.addSearch();

             tempPos = threadFrame->moveDirection(currentPos,direction);

                                        //if the next node is a wall or off the graph
                                        if((threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() == 1)||(!threadFrame-
>pointInGraph(tempPos))){
                                                  //change direction to try and navigate wall
                                           direction++;
                                                  if(direction == 9){
                                                            direction = 1;
                                                  }
                                                  //move in that direction
                                                  tempPos = threadFrame-
>moveDirection(currentPos,direction);
                                        }
                                        //if the next node in the direction is a space
                                        else{
                                                  //move into the space
                                                  tempPos = threadFrame-
>moveDirection(currentPos,direction-1);

                                                  //if the current position is NOT a wall, then we
have found an area to move around it
                                                  if((threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() != 1)&&(threadFrame-
>pointInGraph(tempPos))){ //&& (threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() >= 1) && (threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() <= 3)){


                                                            while((threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() != 1)){
                                                               tempPos = threadFrame-
>moveDirection(currentPos,direction);
                                                                      direction--;

                                                                      if(direction == 0){
                                                                         direction = 8;
                                                                      }
                                                            }
                    direction++;
                                                        if(direction == 9){
                                                           direction = 1;
                                                        }

                                                        direction++;
                                                        if(direction == 9){
                                                           direction = 1;
                                                        }

                    tempPos = threadFrame->moveDirection(currentPos,direction);

                                                        previousPos = currentPos;
                        currentPos = tempPos;

                                                        //if the goal node can be seen from here
                                                        if(threadFrame-
>lineofsight(tempPos.x,tempPos.y)){
                                                           sight = true;
                                                        }
                                                        //or the orginal direction has been achieved
                                                        else if(direction == originalDirection){
                                                                 //return to normal movement
                                                           sight = true;
                                                        }
                                                  }
                                                  //if the node was a wall at the temp move
                                                  else{
                                                        //continue moving in direction to try and
traverse the obstacle
                                                        tempPos = threadFrame-
>moveDirection(currentPos,direction);
                                            previousPos = currentPos;
                        currentPos = tempPos;
                                                  }

                                      dc.SelectObject(&BlueThickPen);
                  threadFrame-
>node_graph[tempPos.x][tempPos.y].setParentNode(previousPos.x,previousPos.y);
                                      //draw a line from the previous node to the destination node
                                      dc.MoveTo(previousPos.x*30+15,previousPos.y*30+15);
                                      dc.LineTo(currentPos.x*30+15,currentPos.y*30+15);
                                                  Sleep(200);

                                                  //if the current node in the search is the end node
                                                  if(threadFrame-
>node_graph[currentPos.x][currentPos.y].get_contents() == 3){
                                                        complete = true;
```

```
                                                algTrack.display(threadFrame-
>drawLine());
                                                threadFrame->iSTATUS = 0;
                                                return 0;
                                        }

                                        algTrack.addStep();
                                }
                        }while(!sight);

                }else{
                        sight = false;
                    previousPos = currentPos;
                currentPos = tempPos;
                        dc.SelectObject(&BlueThickPen);

                        //draw a line from the previous node to the destination node
                        threadFrame-
>node_graph[currentPos.x][currentPos.y].setParentNode(previousPos.x,previousPos.y);
                        dc.MoveTo(previousPos.x*30+15,previousPos.y*30+15);
                        dc.LineTo(currentPos.x*30+15,currentPos.y*30+15);

                        algTrack.addStep();
                }

                //if the algorithm isnt paused
                if(threadFrame->iSTATUS != 0){
                        //if the current node in the search is the end node
                        if(threadFrame-
>node_graph[currentPos.x][currentPos.y].get_contents() == 3){
                      complete = true;
                                algTrack.display(threadFrame->drawLine());
                                threadFrame->iSTATUS = 0;
                                return 0;
                        }
                Sleep(200);
                }
        }while((!complete)&&(threadFrame->iSTATUS != 7)&&(threadFrame-
>iSTATUS != 6));
        threadFrame->iSTATUS = 0;
   return 0;
}

////////////////////////////////////////////////////////////
//The breadth first search algorithm
////////////////////////////////////////////////////////////
UINT breadthFirstSearch(LPVOID pParam){

        //get a pointer to the frame class that this being derived from!!
   CMainFrame* threadFrame = (CMainFrame*)pParam;
```

```
    CClientDC dc(threadFrame);
    CPoint currentPos, previousPos, tempPos;
        CBrush brushBlue(RGB(0, 0, 255));
    bool complete = false;
        int direction = 0;
        CString string;
        CPoint start_node,end_node;

        threadFrame->updateGrid();

        //get the position of the start node
        currentPos = threadFrame->start_node->getPosition();
        start_node = currentPos;
        end_node = threadFrame->end_node->getPosition();
    tempPos = currentPos;

        //create the queue that I will use
        std::queue <CPoint> queue;

        //push the start node onto the queue
        queue.push(currentPos);
        //set the node as being searched
        threadFrame->node_graph[currentPos.x][currentPos.y].setSearched(true);

        //used to track the performance of the algorithm
        performanceTracker algTrack;

        //perform the algorithm while it isnt complete or reset hasnt been pressed
        do{
            //if the algorithm isnt paused
            if(threadFrame->iSTATUS != 0){

                currentPos = queue.front();

        //go through all the directions of movement
                for(int counter = 1;counter <= 8; counter++){

                    algTrack.addSearch();
                    //temporarily move to point
                    tempPos = threadFrame-
>moveDirection(currentPos,counter);

                    //if the node is one the graph and it hasnt already been
searched then push it onto the queue
                    if((threadFrame-
>pointInGraph(tempPos))&&(!threadFrame-
>node_graph[tempPos.x][tempPos.y].getSearched())&&(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() != 1)&&(!complete)){
```

```
                                        algTrack.addStep();

                                        //push the node onto the back of the queue
                                        queue.push(tempPos);
                                        //Set the node as being searched
                                        threadFrame-
>node_graph[tempPos.x][tempPos.y].setSearched(true);
                                        //Set the nodes parent node as the current node
                                        threadFrame-
>node_graph[tempPos.x][tempPos.y].setParentNode(currentPos.x,currentPos.y);

                dc.SelectObject(&BlueThickPen);
                                //draw a line from the previous node to the destination node
                                dc.MoveTo(currentPos.x*30+15,currentPos.y*30+15);
                                dc.LineTo(tempPos.x*30+15,tempPos.y*30+15);

                Sleep(200);

                                    //if the current node in the search is the end node
                                        if(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() == 3){
                                                complete = true;
                                                algTrack.display(threadFrame-
>drawLine());

                                                threadFrame->iSTATUS = 0;
                                                return 0;
                                        }
                                }
                        }
                        queue.pop();
                }
          }while((!complete)&&(threadFrame->iSTATUS != 7)&&(threadFrame-
>iSTATUS != 6));
          threadFrame->iSTATUS = 0;
    return 0;
}
/////////////////////////////////////////////////////
//Dijkstras algorithm
/////////////////////////////////////////////////////
UINT dijkstraSearch(LPVOID pParam){

          //get a pointer to the frame class that this being derived from!!
    CMainFrame* threadFrame = (CMainFrame*)pParam;

    CClientDC dc(threadFrame);
    CPoint currentPos, previousPos, tempPos;
    bool complete = false;
          int direction = 0;
          CString string;
          CPoint start_node,end_node;
```

```
		bool inQueue = false;

		threadFrame->updateGrid();

		//get the position of the start node
		currentPos = threadFrame->start_node->getPosition();
		start_node = currentPos;
		end_node = threadFrame->end_node->getPosition();
	tempPos = currentPos;

		node *ptrCurrentPos;

		//set the node as being searched
		threadFrame->node_graph[currentPos.x][currentPos.y].setSearched(true);

		//create the priority queue and place the first element on it
		priorityQueue priQueue(&threadFrame-
>node_graph[start_node.x][start_node.y]);
		std::vector <node *> closedList;

		//used to track the performance of the algorithm
		performanceTracker algTrack;

		//perform the algorithm while it isnt complete or reset hasnt been pressed
		do{
			//if the algorithm isnt paused
			if(threadFrame->iSTATUS != 0){

				//pop the first node off the queue
				ptrCurrentPos = priQueue.pop();
				currentPos = ptrCurrentPos->getPosition();

				//go through all the directions of movement
				for(int counter = 1;counter <= 8; counter++){

					algTrack.addSearch();

				  //temporarily move to point
					tempPos = threadFrame-
>moveDirection(currentPos,counter);

					//search the list to see if the element is on it
					for(int i = 0; i < closedList.size();i++){
						if(closedList[i] == &threadFrame-
>node_graph[tempPos.x][tempPos.y]){
							inQueue = true;
							i = closedList.size();
						}
						else{
							inQueue = false;
```

```
                                        }
                                }

                                //if the node is one the graph and it hasnt already been
searched then push it onto the queue
                                if((!inQueue)&&(threadFrame-
>pointInGraph(tempPos))&&(!threadFrame-
>node_graph[tempPos.x][tempPos.y].getSearched())&&(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() != 1)&&(!complete)){

                                        algTrack.addStep();

                                        //Set the node as being searched
                                        threadFrame-
>node_graph[tempPos.x][tempPos.y].setSearched(true);
                                        //Set the nodes parent node as the current node
                                        threadFrame-
>node_graph[tempPos.x][tempPos.y].setParentNode(currentPos.x,currentPos.y);


                                        //set the distance if diagonal the distance is greater
                                        if(counter %2 != 0){
                                                threadFrame-
>node_graph[tempPos.x][tempPos.y].setHeuristic(1.0,threadFrame-
>node_graph[currentPos.x][currentPos.y].getHeuristic());
                                        }else{
                                            threadFrame-
>node_graph[tempPos.x][tempPos.y].setHeuristic(1.4,threadFrame-
>node_graph[currentPos.x][currentPos.y].getHeuristic());
                                        }

                                        //push the node onto the queue which will
prioritise it
                                        priQueue.push(&threadFrame-
>node_graph[tempPos.x][tempPos.y]);


                dc.SelectObject(&BlueThickPen);
                                //draw a line from the previous node to the destination node
                                dc.MoveTo(currentPos.x*30+15,currentPos.y*30+15);
                                dc.LineTo(tempPos.x*30+15,tempPos.y*30+15);

                Sleep(200);

                                    //if the current node in the search is the end node
                                        if(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() == 3){
                                                complete = true;
                                                algTrack.display(threadFrame-
>drawLine());
```

```
                                        threadFrame->iSTATUS = 0;
                                        return 0;
                                    }
                                }
                            }

                        }
            }while((!complete)&&(threadFrame->iSTATUS != 7)&&(threadFrame-
>iSTATUS != 6));
            threadFrame->iSTATUS = 0;
    return 0;
}

//////////////////////////////////////////////////
//The best first search algorithm
//////////////////////////////////////////////////
UINT bestFirstSearch(LPVOID pParam){

            //get a pointer to the frame class that this being derived from!!
    CMainFrame* threadFrame = (CMainFrame*)pParam;

    CClientDC dc(threadFrame);
    CPoint currentPos, previousPos, tempPos;
            CBrush brushBlue(RGB(0, 0, 255));
    bool complete = false;
            int direction = 0;
            CString string;
            CPoint start_node,end_node;

            threadFrame->updateGrid();

            //get the position of the start node
            currentPos = threadFrame->start_node->getPosition();
            start_node = currentPos;
            end_node = threadFrame->end_node->getPosition();
    tempPos = currentPos;

            node *ptrCurrentPos;

            //set the node as being searched
            threadFrame->node_graph[currentPos.x][currentPos.y].setSearched(true);

            //create the priority queue and place the first element on it
            priorityQueue priQueue(&threadFrame-
>node_graph[start_node.x][start_node.y]);

            //used to track the performance of the algorithm
            performanceTracker algTrack;

            //perform the algorithm while it isnt complete or reset hasnt been pressed
```

```
		do{
			//if the algorithm isnt paused
			if(threadFrame->iSTATUS != 0){

				//pop the first node off the queue
				ptrCurrentPos = priQueue.pop();
				currentPos = ptrCurrentPos->getPosition();

				//go through all the directions of movement
				for(int counter = 1;counter <= 8; counter++){

					algTrack.addSearch();
				    //temporarily move to point
					tempPos = threadFrame-
>moveDirection(currentPos,counter);

					//if the node is one the graph and it hasnt already been
searched then push it onto the queue
					if((threadFrame-
>pointInGraph(tempPos))&&(!threadFrame-
>node_graph[tempPos.x][tempPos.y].getSearched())&&(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() != 1)&&(!complete)){

						algTrack.addStep();

						//Set the node as being searched
						threadFrame-
>node_graph[tempPos.x][tempPos.y].setSearched(true);
						//Set the nodes parent node as the current node
						threadFrame-
>node_graph[tempPos.x][tempPos.y].setParentNode(currentPos.x,currentPos.y);


						//set the heuristic for the node
						threadFrame-
>node_graph[tempPos.x][tempPos.y].setHeuristic(threadFrame-
>calcHeuristic(tempPos,end_node));

						//push the node onto the queue which will
prioritise it
						priQueue.push(&threadFrame-
>node_graph[tempPos.x][tempPos.y]);

		dc.SelectObject(&BlueThickPen);
				//draw a line from the previous node to the destination node
				dc.MoveTo(currentPos.x*30+15,currentPos.y*30+15);
				dc.LineTo(tempPos.x*30+15,tempPos.y*30+15);

		Sleep(200);
```

```
                                        //if the current node in the search is the end node
                                            if(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() == 3){
                                                complete = true;
                                                algTrack.display(threadFrame-
>drawLine());

                                                threadFrame->iSTATUS = 0;
                                                return 0;
                                            }
                                        }
                                }
                        }
        }while((!complete)&&(threadFrame->iSTATUS != 7)&&(threadFrame-
>iSTATUS != 6));

        threadFrame->iSTATUS = 0;
  return 0;
}
```

A-Star Algorithm within mainfrm.cpp

```
////////////////////////////////////////////////
//The A-Star search algorithm
////////////////////////////////////////////////
UINT aStar(LPVOID pParam){

        //get a pointer to the frame class that this being derived from!!
   CMainFrame* threadFrame = (CMainFrame*)pParam;

   CClientDC dc(threadFrame);
   CPoint currentPos, previousPos, tempPos;
        CBrush brushBlue(RGB(0, 0, 255));
   bool complete = false, inQueue = false;
        int direction = 0;
        CString string;
        CPoint start_node,end_node;

        threadFrame->updateGrid();

        //get the position of the start node
        currentPos = threadFrame->start_node->getPosition();
        start_node = currentPos;
        end_node = threadFrame->end_node->getPosition();
   tempPos = currentPos;

        node *ptrCurrentPos;

        //set the node as being searched
        threadFrame->node_graph[currentPos.x][currentPos.y].setSearched(true);
        threadFrame->node_graph[currentPos.x][currentPos.y].setDistance(0,0);

        //create the priority queue and place the first element on it
        priorityQueue priQueue(&threadFrame-
>node_graph[start_node.x][start_node.y]);
        std::vector <node *> closedList;

   //used to track the performance of the algorithm
        performanceTracker algTrack;

        //perform the algorithm while it isnt complete or reset hasnt been pressed
        do{
               //if the algorithm isnt paused
               if(threadFrame->iSTATUS != 0){

                      //pop the first node off the queue
                      ptrCurrentPos = priQueue.pop();
                      currentPos = ptrCurrentPos->getPosition();
```

```
                              //go through all the directions of movement
                              for(int counter = 1;counter <= 8; counter++){
                                        algTrack.addSearch();
                                 //temporarily move to point
                                        tempPos = threadFrame-
>moveDirection(currentPos,counter);

                                        //search the closed list to see if the element is on it
                                        for(int i = 0; i < closedList.size();i++){
                                                  if(closedList[i] == &threadFrame-
>node_graph[tempPos.x][tempPos.y]){
                                                            inQueue = true;
                                                            i = closedList.size();
                                                  }
                                                  else{
                                                            inQueue = false;
                                                  }
                                        }

                                        //if the node is one the graph and it hasnt already been
searched then push it onto the queue
                                        if((threadFrame-
>pointInGraph(tempPos))&&(!inQueue)&&(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() != 1)&&(!complete)){

                                                  float tempHeuristic = 0;
                                                  float tempDistance = 0;

                                                  //set the distance if diagonal the distance is greater
                                                  if(counter %2 != 0){
                                                            tempDistance = 1.0 + threadFrame-
>node_graph[currentPos.x][currentPos.y].getDistance();
                                                  }else{
                                                            tempDistance = 1.4 + threadFrame-
>node_graph[currentPos.x][currentPos.y].getDistance();
                                                  }

                                                  tempHeuristic = threadFrame-
>calcHeuristic(tempPos,end_node) + tempDistance;

                                                  //if the node has already been searched and the
heuristic is lower then we need to redraw over it
                                                  if((threadFrame-
>node_graph[tempPos.x][tempPos.y].getSearched())&&(threadFrame-
>node_graph[tempPos.x][tempPos.y].getHeuristic() > tempHeuristic)){
                                                            dc.SelectObject(&BlackThinPen);

dc.Rectangle(tempPos.x*30,tempPos.y*30,(tempPos.x*30)+30,(tempPos.y*30)+30);
                                                            threadFrame-
>node_graph[tempPos.x][tempPos.y].setParentNode(currentPos.x,currentPos.y);
```

```
                                                //set the heuristic for the node
                                                threadFrame-
>node_graph[tempPos.x][tempPos.y].setHeuristic(tempHeuristic);
                                                threadFrame-
>node_graph[tempPos.x][tempPos.y].setDistance(tempDistance);
                                                //The item will need repositioning on the
queue
                                                priQueue.rePosition(&threadFrame-
>node_graph[tempPos.x][tempPos.y]);
                                                dc.SelectObject(&BlueThickPen);
                                                //draw a line from the previous node to the
destination node

        dc.MoveTo(currentPos.x*30+15,currentPos.y*30+15);

        dc.LineTo(tempPos.x*30+15,tempPos.y*30+15);
                Sleep(200);
                                                algTrack.addStep();

                                        }else if(threadFrame-
>node_graph[tempPos.x][tempPos.y].getSearched() == false){
                                            //Set the node as being searched
                                                threadFrame-
>node_graph[tempPos.x][tempPos.y].setSearched(true);
                                                //Set the distance
                                                threadFrame-
>node_graph[tempPos.x][tempPos.y].setDistance(tempDistance);
                                                //Set the nodes parent node as the current
node
                                      threadFrame-
>node_graph[tempPos.x][tempPos.y].setParentNode(currentPos.x,currentPos.y);

                                                //set the heuristic for the node
                                                threadFrame-
>node_graph[tempPos.x][tempPos.y].setHeuristic(tempHeuristic);
                                                //push the node onto the queue which will
prioritise it
                                                priQueue.push(&threadFrame-
>node_graph[tempPos.x][tempPos.y]);
                                                dc.SelectObject(&BlueThickPen);
                                                //draw a line from the previous node to the
destination node

        dc.MoveTo(currentPos.x*30+15,currentPos.y*30+15);

        dc.LineTo(tempPos.x*30+15,tempPos.y*30+15);
                                                Sleep(200);
                                                algTrack.addStep();
                                        }
                                        //if the current node in the search is the end node
                                                if(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() == 3){
                                                        complete = true;
                                                        algTrack.display(threadFrame-
>drawLine());

                                                        threadFrame->iSTATUS = 0;
                                                        return 0;
                                                }
                                        }
        //push the node onto the finished queue
                                closedList.push_back(ptrCurrentPos);

                        }

                }
        }while((!complete)&&(threadFrame->iSTATUS != 7)&&(threadFrame-
>iSTATUS != 6));
        threadFrame->iSTATUS = 0;
   return 0;
}
```

Dynamic A* (D*) Algorithm within mainfrm.cpp

```
/////////////////////////////////////////////////
//The Dynamic A* Algorithm
/////////////////////////////////////////////////
UINT dynamicAstar(LPVOID pParam){

        //get a pointer to the frame class that this being derived from!!
   CMainFrame* threadFrame = (CMainFrame*)pParam;

   CClientDC dc(threadFrame);
   CPoint currentPos, previousPos, tempPos;
        bool complete = false, inQueue = false, draw = false,change = false;
        int direction = 0,shortest=0,x,y;
        CString string;
        CPoint start_node,end_node,shortest_move;

        threadFrame->updateGrid();
        CRect circle;

        for(x = 0; x <= 20; x++){
               for(y = 0; y <= 20; y++){
                      threadFrame->node_graph[x][y].setSearched(false);
               }
        }

        //get the position of the start node
        //with this algorithm we are starting our search from the end node and trying to
find the start node
        currentPos = threadFrame->end_node->getPosition();
        start_node = currentPos;

        end_node = threadFrame->start_node->getPosition();
   tempPos = currentPos;

        //temporary pointer to a node
        node *ptrCurrentPos;

        //set the node as being searched
        threadFrame->node_graph[currentPos.x][currentPos.y].setSearched(true);

        //set the start nodes distance as 0
        threadFrame->node_graph[currentPos.x][currentPos.y].setDistance(0);

        //create the priority queue and place the first element on it
        priorityQueue priQueue(&threadFrame-
>node_graph[start_node.x][start_node.y]);
        std::vector <node *> closedList;
```

```
		//used to track the performance of the algorithm
		performanceTracker algTrack;

		//perform the algorithm while it isnt complete or reset hasnt been pressed
		do{
			//if the algorithm isnt paused
			if(threadFrame->iSTATUS != 0){

				//pop the first node off the queue
				ptrCurrentPos = priQueue.pop();
				currentPos = ptrCurrentPos->getPosition();

				//go through all the directions of movement
				for(int counter = 1;counter <= 8; counter++){

					//temporarily move to point to check if it is ok
					tempPos = threadFrame-
>moveDirection(currentPos,counter);

					if((threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents()!=1)&&(threadFrame-
>pointInGraph(tempPos))){
						algTrack.addSearch();


						float tempDistance = 0;

						//set the distance if diagonal the distance is greater
						if(counter %2 != 0){
							tempDistance = 1.0 + threadFrame-
>node_graph[currentPos.x][currentPos.y].getDistance();
						}else{
							tempDistance = 1.2 + threadFrame-
>node_graph[currentPos.x][currentPos.y].getDistance();
						}

						//if the node is new and hasnt been searched
generate its heuristic
						if((threadFrame-
>node_graph[tempPos.x][tempPos.y].get_state() == 0)&&(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents()!=3)){
							//Set the node as being on the open queue
							threadFrame-
>node_graph[tempPos.x][tempPos.y].set_state(1);
							//set the distance from this node
							threadFrame-
>node_graph[tempPos.x][tempPos.y].setDistance(tempDistance);
							//Set the nodes parent node as the current
node
```

```
						threadFrame-
>node_graph[tempPos.x][tempPos.y].setParentNode(currentPos.x,currentPos.y);

						//set the heuristic for the node
						threadFrame-
>node_graph[tempPos.x][tempPos.y].setHeuristic(tempDistance);
						//Push the new item onto the queue
						priQueue.push(&threadFrame-
>node_graph[tempPos.x][tempPos.y]);
						//add a step to the performance manager
						algTrack.addStep();
						draw = true;
						change = false;
						threadFrame-
>node_graph[tempPos.x][tempPos.y].setSearched(true);
						Sleep(200);
					}
					//if its heuristic is lower than the one already
assigned to the node.
					else if((threadFrame-
>node_graph[tempPos.x][tempPos.y].get_state() != 0) && (threadFrame-
>node_graph[tempPos.x][tempPos.y].getHeuristic() > tempDistance)&&(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents()!=3)){
						//set the distance from this node
						threadFrame-
>node_graph[tempPos.x][tempPos.y].setDistance(tempDistance);
						//Set the nodes parent node as the current
node
						threadFrame-
>node_graph[tempPos.x][tempPos.y].setParentNode(currentPos.x,currentPos.y);

						//set the heuristic for the node
						threadFrame-
>node_graph[tempPos.x][tempPos.y].setHeuristic(tempDistance);

						//push the item back onto the queue
						if(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_state() == 2)
							priQueue.push(&threadFrame-
>node_graph[tempPos.x][tempPos.y]);
						else

	priQueue.rePosition(&threadFrame->node_graph[tempPos.x][tempPos.y]);

						dc.SelectObject(&brushWhite);
						dc.SelectObject(&BlackThinPen);

	dc.Rectangle(tempPos.x*30,tempPos.y*30,(tempPos.x*30)+30,(tempPos.y*30)
+30);
```

```
                                                //if the current node is the start node then
we redraw the start point
                                                if(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() == 2){
                                                        dc.SelectObject(&BlackThickPen);
                                                        dc.SelectObject(&brushGreen);

        circle.SetRect(x*30,y*30,x*30+30,y*30+30);
                                                        dc.Ellipse(&circle);
                                                }

                                                draw = true;
                                                change = true;
                                                //threadFrame-
>MessageBox(_T("CHANGE!!!"));
                                                threadFrame-
>node_graph[tempPos.x][tempPos.y].setSearched(true);
                                                Sleep(200);

                                        }else if((threadFrame-
>node_graph[tempPos.x][tempPos.y].get_state() == 2)&&(tempDistance =
threadFrame->node_graph[tempPos.x][tempPos.y].getDistance())&&(!threadFrame-
>node_graph[tempPos.x][tempPos.y].getSearched())){
                                                //Set the node as being on the open queue
                                                threadFrame-
>node_graph[tempPos.x][tempPos.y].set_state(1);
                                                //Push the new item onto the queue
                                                priQueue.push(&threadFrame-
>node_graph[tempPos.x][tempPos.y]);
                                                threadFrame-
>node_graph[tempPos.x][tempPos.y].setSearched(true);
                                                draw = true;
                                        }

                                        if((threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents()!=3)&&(draw)){

                                                CPoint arrow[4];

                                                shortest = counter - 4;

                                                //get the coordinates for the drawing of the
arrows
                                                switch(shortest){
                                                        case 1:{
                                                                //threadFrame-
>MessageBox(_T("CASE 1"));
                                                                arrow[0] =
CPoint(5+(tempPos.x*30),15+(tempPos.y*30));
```

```
                                        arrow[1] =
CPoint(15+(tempPos.x*30),5+(tempPos.y*30));
                                        arrow[2] =
CPoint(25+(tempPos.x*30),15+(tempPos.y*30));
                                        arrow[3] =
CPoint(5+(tempPos.x*30),15+(tempPos.y*30));
                                    }break;
                                    case 2:{
                                        //threadFrame-
>MessageBox(_T("CASE 2"));
                                        arrow[0] =
CPoint(5+(tempPos.x*30),5+(tempPos.y*30));
                                        arrow[1] =
CPoint(25+(tempPos.x*30),5+(tempPos.y*30));
                                        arrow[2] =
CPoint(25+(tempPos.x*30),25+(tempPos.y*30));
                                        arrow[3] =
CPoint(5+(tempPos.x*30),5+(tempPos.y*30));
                                    }break;
                                    case 3:{
                                        //threadFrame-
>MessageBox(_T("CASE 3"));
                                        arrow[0] =
CPoint(15+(tempPos.x*30),5+(tempPos.y*30));
                                        arrow[1] =
CPoint(25+(tempPos.x*30),15+(tempPos.y*30));
                                        arrow[2] =
CPoint(15+(tempPos.x*30),25+(tempPos.y*30));
                                        arrow[3] =
CPoint(15+(tempPos.x*30),5+(tempPos.y*30));
                                    }break;
                                    case 4:{
                                        //threadFrame-
>MessageBox(_T("CASE 4"));
                                        arrow[0] =
CPoint(25+(tempPos.x*30),5+(tempPos.y*30));
                                        arrow[1] =
CPoint(25+(tempPos.x*30),25+(tempPos.y*30));
                                        arrow[2] =
CPoint(5+(tempPos.x*30),25+(tempPos.y*30));
                                        arrow[3] =
CPoint(25+(tempPos.x*30),5+(tempPos.y*30));
                                    }break;
                                    case -3:{
                                        //threadFrame-
>MessageBox(_T("CASE 5"));
                                        arrow[0] =
CPoint(5+(tempPos.x*30),15+(tempPos.y*30));
                                        arrow[1] =
CPoint(25+(tempPos.x*30),15+(tempPos.y*30));
```

```
                                        arrow[2] =
CPoint(15+(tempPos.x*30),25+(tempPos.y*30));
                                        arrow[3] =
CPoint(5+(tempPos.x*30),15+(tempPos.y*30));
                              }break;
                              case -2:{
                                        //threadFrame-
>MessageBox(_T("CASE 6"));
                                        arrow[0] =
CPoint(5+(tempPos.x*30),5+(tempPos.y*30));
                                        arrow[1] =
CPoint(25+(tempPos.x*30),25+(tempPos.y*30));
                                        arrow[2] =
CPoint(5+(tempPos.x*30),25+(tempPos.y*30));
                                        arrow[3] =
CPoint(5+(tempPos.x*30),5+(tempPos.y*30));
                              }break;
                              case -1:{
                                        //threadFrame-
>MessageBox(_T("CASE 7"));
                                        arrow[0] =
CPoint(15+(tempPos.x*30),5+(tempPos.y*30));
                                        arrow[1] =
CPoint(15+(tempPos.x*30),25+(tempPos.y*30));
                                        arrow[2] =
CPoint(5+(tempPos.x*30),15+(tempPos.y*30));
                                        arrow[3] =
CPoint(15+(tempPos.x*30),5+(tempPos.y*30));
                              }
                              break;
                              case 0:{
                                        //threadFrame-
>MessageBox(_T("CASE 8"));
                                        arrow[0] =
CPoint(25+(tempPos.x*30),5+(tempPos.y*30));
                                        arrow[1] =
CPoint(5+(tempPos.x*30),25+(tempPos.y*30));
                                        arrow[2] =
CPoint(5+(tempPos.x*30),5+(tempPos.y*30));
                                        arrow[3] =
CPoint(25+(tempPos.x*30),5+(tempPos.y*30));
                              }break;
                          }

          if(change){
                              change = false;
                              dc.SelectObject(&RedThickPen);
                          }
                          else
                              dc.SelectObject(&BlueThickPen);
```

```
                                                //draw the arrow for the shortest node here
                                                dc.Polyline(arrow,4);

                                                draw = false;
                                        }
                                }
                                //set the node as being complete
                                threadFrame-
>node_graph[currentPos.x][currentPos.y].set_state(2);
                        }
                }

                //if the priority queue is empty
                if(priQueue.isEmpty()){
                        complete = true;
                        //algTrack.display(threadFrame->drawLine());
                        threadFrame->MessageBox(_T("Complete"));
                        threadFrame->iSTATUS = 0;
                        return 0;
                }

        }while((!complete)&&(threadFrame->iSTATUS != 7)&&(threadFrame-
>iSTATUS != 6));
        threadFrame->iSTATUS = 0;
   return 0;
}
```

Pathjoiner Algorithm within mainfrm.cpp

```
/////////////////////////////////////////////////////
//The environment addition pathfinding algorithm
/////////////////////////////////////////////////////
UINT environmentSubtraction(LPVOID pParam){

        //get a pointer to the frame class that this being derived from!!
   CMainFrame* threadFrame = (CMainFrame*)pParam;

        //if there has been no path generated by a previous algorithm then we cant
change the path
        if(!threadFrame->pathGenerated()){
                threadFrame->MessageBox(_T("No previous path to search"));
           return 0;
        }

   bool nodeCleared = false;

        CClientDC dc(threadFrame);
   CPoint currentPos, previousPos, tempPos;
   bool complete = false;
        int direction = 0;
        CString string;
        CPoint start_node,end_node;
        bool inQueue = false;

        threadFrame->updateGrid();

        //do a simple reset of the graph
        threadFrame->simpleResetGraph();

        //get the position of the changed node
        currentPos = threadFrame->change_node->getPosition();
        start_node = currentPos;
        end_node = threadFrame->end_node->getPosition();
        tempPos = currentPos;

        //pointer to the current position and the pointer to the reference node.
        node *ptrCurrentPos,*ptrReferenceNode = NULL;

        //distance to the reference node
   float referenceNodeDistance;
        CPoint referenceNodeParent;

        //set the node as being searched
        threadFrame->node_graph[currentPos.x][currentPos.y].setSearched(true);

        //create the priority queue and place the first element on it
        priorityQueue priQueue(&threadFrame-
>node_graph[start_node.x][start_node.y]);

        //set the distance of the first node to 0]
        threadFrame->node_graph[start_node.x][start_node.y].setDistance(0);

        //used to track the performance of the algorithm
        performanceTracker algTrack;

        //perform the algorithm while it isnt complete or reset hasnt been pressed
        do{
                //if the algorithm isnt paused
                if(threadFrame->iSTATUS != 0){

                        //pop the first node off the queue
                        ptrCurrentPos = priQueue.pop();
                        currentPos = ptrCurrentPos->getPosition();

                        //go through all the directions of movement
                        for(int counter = 1;counter <= 8; counter++){

                                algTrack.addSearch();

                                //temporarily move to point
                                tempPos = threadFrame-
>moveDirection(currentPos,counter);


                                //if the node is one the graph and it hasnt already been
searched then push it onto the queue
                                if((!threadFrame-
>node_graph[tempPos.x][tempPos.y].onShortestPath())&&(threadFrame-
>pointInGraph(tempPos))&&(!threadFrame-
>node_graph[tempPos.x][tempPos.y].getSearched())&&(threadFrame-
>node_graph[tempPos.x][tempPos.y].get_contents() != 1)&&(!complete)){

                                        algTrack.addStep();

                                        //Set the node as being searched
                                        threadFrame-
>node_graph[tempPos.x][tempPos.y].setSearched(true);
                                        //Set the nodes parent node as the current node
                                        threadFrame-
>node_graph[tempPos.x][tempPos.y].setParentNode(currentPos.x,currentPos.y);


                                        //set the distance if diagonal the distance is greater
                                        if(counter %2 != 0){
```

```
                                        threadFrame-
>node_graph[tempPos.x][tempPos.y].setHeuristic(1.0,threadFrame-
>node_graph[currentPos.x][currentPos.y].getHeuristic());
                                }else{
                                        threadFrame-
>node_graph[tempPos.x][tempPos.y].setHeuristic(1.4,threadFrame-
>node_graph[currentPos.x][currentPos.y].getHeuristic());
                                }

                                //push the node onto the queue which will
prioritise it
                                priQueue.push(&threadFrame-
>node_graph[tempPos.x][tempPos.y]);


                                dc.SelectObject(&BlueThickPen);
                                //draw a line from the previous node to the
destination node

        dc.MoveTo(currentPos.x*30+15,currentPos.y*30+15);
                                dc.LineTo(tempPos.x*30+15,tempPos.y*30+15);

                                Sleep(200);

                        //if the node we have found is on the shortest path then we
have to examine it
                        }else if((threadFrame-
>node_graph[tempPos.x][tempPos.y].onShortestPath())&&(!threadFrame-
>node_graph[tempPos.x][tempPos.y].getSearched())){

                                //tempDistance stores a temporary distance
            float tempDistance;

                                //Set the node as being searched
                                threadFrame-
>node_graph[tempPos.x][tempPos.y].setSearched(true);

                                //calculate the temporary distance adjusting for a
diagonal movement
                                if(counter %2 != 0){
                                        tempDistance = 1.0 + threadFrame-
>node_graph[currentPos.x][currentPos.y].getHeuristic();
                                }else{
                                        tempDistance = 1.4 + threadFrame-
>node_graph[currentPos.x][currentPos.y].getHeuristic();
                                }

                                //if the node is the first node we find on the
shortest path, this becomes our reference node
                                if(ptrReferenceNode == NULL){
                                        ptrReferenceNode = &threadFrame-
>node_graph[tempPos.x][tempPos.y];
                                        referenceNodeDistance = tempDistance;
                                        referenceNodeParent = currentPos;

                                        dc.SelectObject(&BlueThickPen);
                                        //draw a line from the previous node to the
destination node

    dc.MoveTo(currentPos.x*30+15,currentPos.y*30+15);

    dc.LineTo(tempPos.x*30+15,tempPos.y*30+15);

                                        Sleep(200);
                                    }
                                    else{

                                        float nodeDistance;

                                        //if the distance of the reference node is
greater than the tested node
                                        // and calculate the distance between the 2
nodes
                                        if((ptrReferenceNode-
>getDistance()>threadFrame->node_graph[tempPos.x][tempPos.y].getDistance()))
                                            nodeDistance = ptrReferenceNode-
>getDistance() - threadFrame->node_graph[tempPos.x][tempPos.y].getDistance();
                                        else
                                            nodeDistance = threadFrame-
>node_graph[tempPos.x][tempPos.y].getDistance() - ptrReferenceNode->getDistance();



                                        dc.SelectObject(&BlueThickPen);
                                        //draw a line from the previous node to the
destination node

    dc.MoveTo(currentPos.x*30+15,currentPos.y*30+15);

    dc.LineTo(tempPos.x*30+15,tempPos.y*30+15);

                                        Sleep(200);

                                        //if the distance between the two nodes on
the shortest path is greater than the sum of
                                        //the distances of the 2 nodes from the
orginal starting node then we have a shorter path
                                        //IF((X1 + X2)< Y)
                                        if((referenceNodeDistance +
tempDistance) < nodeDistance ){
```

```
                                        CPoint tempParent,previousParent;

                                        //now that we have found a shorter
path we have to amend the original path to incorporate it
                                        //find the lower if the 2 distances so
that we can amend the nodes parents properly
                                        if((ptrReferenceNode-
>getDistance()>threadFrame->node_graph[tempPos.x][tempPos.y].getDistance())){
                                                //store the original parent of
the node
                                                previousParent =
ptrCurrentPos->getParentNode();;
                                                ptrCurrentPos =
&threadFrame->node_graph[tempPos.x][tempPos.y];
                                                tempParent = currentPos;
                                                threadFrame-
>node_graph[currentPos.x][currentPos.y].setParentNode(tempPos.x,tempPos.y);
                                                ptrReferenceNode-
>setParentNode(referenceNodeParent.x,referenceNodeParent.y);
                                        }else{
                                                //threadFrame-
>MessageBox(_T("Reference lower"));
                                                //store the original parent of
the node
                                                previousParent =
threadFrame-
>node_graph[referenceNodeParent.x][referenceNodeParent.y].getPosition();
                                                threadFrame-
>node_graph[tempPos.x][tempPos.y].setParentNode(currentPos.x,currentPos.y);
                                            ptrCurrentPos =
ptrReferenceNode;
                                                tempParent =
referenceNodeParent;
                                                threadFrame-
>node_graph[tempParent.x][tempParent.y].setParentNode(ptrReferenceNode-
>getPosition().x,ptrReferenceNode->getPosition().y);

                                        }

                                        //repeat while the parent node is not
equal to the node where the change took place
                                        do{
                                                //the current pointer points
to the tempparent
                                                ptrCurrentPos =
&threadFrame->node_graph[tempParent.x][tempParent.y];

                                                //the tempparent is set to the
previous parent
```

```
                                                  tempParent =
previousParent;

                                                  //the previous parent is now
assigned the parent node of the unaltered node
                                                  previousParent =
threadFrame->node_graph[tempParent.x][tempParent.y].getParentNode();

                                                  //copy the currentposition to
the parent node of the previousparentnode node.
                                                  threadFrame-
>node_graph[tempParent.x][tempParent.y].setParentNode(ptrCurrentPos-
>getPosition().x,ptrCurrentPos->getPosition().y);


        }while((tempParent.x!=start_node.x)||(tempParent.y!=start_node.y));

                                             threadFrame-
>MessageBox(_T("Shorter Path found"));

                                             threadFrame->drawLine();
                                             threadFrame->drawLine2();
                                             return(0);


                                        }
                                        //if the sum of the two distances from the
start node is greater than the length of the entire path at any point quit out
                                        else if((referenceNodeDistance +
tempDistance)>threadFrame->node_graph[end_node.x][end_node.y].getDistance()){
                                             threadFrame-
>MessageBox(_T("No shorter path available"));

                                             threadFrame->iSTATUS = 0;
                                             return 0;
                                        }
                                   }
                              }
                         }
                    }

                    //if the priority queue is empty
                    if(priQueue.isEmpty()){
                         threadFrame->MessageBox(_T("Complete"));
                         threadFrame->iSTATUS = 0;
                         return 0;
                    }

        }while((!complete)&&(threadFrame->iSTATUS != 7)&&(threadFrame-
>iSTATUS != 6));
        threadFrame->iSTATUS = 0;
        return 0;
```

```
}
```

Node.h

```
////////////////////////////////////////////////////
// node.h : interface of the node class
////////////////////////////////////////////////////

#pragma once

class node
{
private:
		int nodeState;

		//private variables
private:
		//what type of node this is
		int contents;
		//various position pointers
		CPoint nodePosition,parentNode;
		//is the node searched
		bool searched;
		//the heuristic of the node
		float heuristic;
		//the distance of the node
		float distance;
		int directionmoved;
		bool ShortestPath;

public:
		node(void);
		node(CPoint position);

		//set and get functions
		void set_contents(int contents);
		int get_state();
		void set_state(int state);
		int get_contents();
		int setIndex()const;
		void setindex(int newindex);
		CPoint getPosition();
		void setParentNode(int x, int y);
		CPoint getParentNode();
		void setPosition(int x, int y);
		bool getSearched();
		void setSearched(bool search);
		void clearNode();
		void setHeuristic(float tempTeuristic);
   void setHeuristic(float tempHeuristic,float parentHeuristic);
		void setDistance(float tempDistance);
```

```
        float getHeuristic();
        void setDistance(float tempDistance, float parentDistance);
        float getDistance();
        void onShortestPath(bool onPath);
        bool onShortestPath();

public:
        virtual ~node(void);
};
```

Node.cpp

```
////////////////////////////////////////////////////////////
// node.cpp : implementation of the node class
////////////////////////////////////////////////////////////

#include "StdAfx.h"
#include "node.h"


//states used for D*
#define NEW 0
#define OPEN 1
#define CLOSED 2


////////////////////////////////////////////////////////
//Node constructor
////////////////////////////////////////////////////////
node::node(void)
{
	contents = 0;
	searched = false;
}

////////////////////////////////////////////////////////
//overloaded node constructor
////////////////////////////////////////////////////////
node::node(CPoint position){
   nodePosition = position;
	parentNode = nodePosition;
	contents = 0;
	searched = false;
	nodeState = NEW;
}

node::~node(void)
{
}

////////////////////////////////////////////////////////
//get the state of this node
////////////////////////////////////////////////////////
int node::get_state(){
   return nodeState;
}
```

```
//////////////////////////////////////////////////
//set the state of this node
//////////////////////////////////////////////////
void node::set_state(int state){
   nodeState = state;
}

//////////////////////////////////////////////////
//get the contents of this node
//////////////////////////////////////////////////
int node::get_contents(){
   return contents;
}



//////////////////////////////////////////////////
//set the conetents of this node
//////////////////////////////////////////////////
void node::set_contents(int newcontents){
   contents = newcontents;
}

//////////////////////////////////////////////////
//get the positon of this node on the graph
//////////////////////////////////////////////////
CPoint node::getPosition(){
   return nodePosition;
}

//////////////////////////////////////////////////
//set the position of this node on the graph
//////////////////////////////////////////////////
void node::setPosition(int x, int y){
  nodePosition.x = x;
  nodePosition.y = y;
}

//////////////////////////////////////////////////
//get the parent node of this node
//////////////////////////////////////////////////
CPoint node::getParentNode(){
   return parentNode;
}
```

```
/////////////////////////////////////////////////////
//set the parent node of this node
/////////////////////////////////////////////////////
void node::setParentNode(int x, int y){
   parentNode.x = x;
         parentNode.y = y;
}

/////////////////////////////////////////////////////
//check if the node is searched
/////////////////////////////////////////////////////
bool node::getSearched(){
   return searched;
}

/////////////////////////////////////////////////////
//set the node to being searched or not
/////////////////////////////////////////////////////
void node::setSearched(bool search){
   searched = search;
}


/////////////////////////////////////////////////////
//Reset the node back to default
/////////////////////////////////////////////////////
void node::clearNode(){
   contents = 0;
         parentNode = nodePosition;
         searched = false;
         heuristic = 0.0;
}

/////////////////////////////////////////////////////
//overloaded setting of the heuristic
/////////////////////////////////////////////////////
void node::setHeuristic(float tempHeuristic,float parentHeuristic){
   heuristic = tempHeuristic + parentHeuristic;
}

/////////////////////////////////////////////////////
//set the heuristic of the node
/////////////////////////////////////////////////////
void node::setHeuristic(float tempHeuristic){
   heuristic = tempHeuristic;
}
```

```
/////////////////////////////////////////////////////
//get the heuristic of the node
/////////////////////////////////////////////////////
float node::getHeuristic(){
   return heuristic;
}

/////////////////////////////////////////////////////
//overloaded setting of the distance
/////////////////////////////////////////////////////
void node::setDistance(float tempDistance, float parentDistance){
   distance = tempDistance + parentDistance;
}

/////////////////////////////////////////////////////
//get the distance of the node
/////////////////////////////////////////////////////
float node::getDistance(){
   return distance;
}

/////////////////////////////////////////////////////
//set the distance of the node
/////////////////////////////////////////////////////
void node::setDistance(float tempDistance){
   distance = tempDistance;
}

/////////////////////////////////////////////////////
//set if the node is on the shortest path or not
/////////////////////////////////////////////////////
void node::onShortestPath(bool onPath){
         ShortestPath = onPath;
}

/////////////////////////////////////////////////////
//check if the node is on the shortest path
/////////////////////////////////////////////////////
bool node::onShortestPath(){
   return ShortestPath;
}
```

priorityQueue.h

```
///////////////////////////////////////////////////
// priorityQueue.h : interface of the priorityQueue class
///////////////////////////////////////////////////

#pragma once
#include "node.h"
#include <queue>

class priorityQueue
{

private:
	node *firstElement,*lastElement;
	int size;

public:
	priorityQueue(void);
	priorityQueue(node *element);

	bool isEmpty();
	int sizeOf();
	void push(node *element);
	node* pop();
	void move(int position,node *element);
	void display();
	void rePosition(node *element);

public:
	~priorityQueue(void);
};
```

priorityQueue.cpp

```
//////////////////////////////////////////////////////
// priorityQueue.cpp : implementation of the priorityQueue class
//////////////////////////////////////////////////////

#include "StdAfx.h"
#include "priorityQueue.h"
#include "node.h"
#include "windows.h"
#include <vector>

//what is to be stored upon the queue


//the vector used to store the nodes
node *queue[200];

priorityQueue::priorityQueue(void)
{
}

priorityQueue::~priorityQueue(void)
{
}

priorityQueue::priorityQueue(node *element){

        CString test;
   CPoint Itest;

        Itest = element->getPosition();
        size = 0;

   firstElement = element;
        lastElement = element;

        queue[0] = element;

        size++;
}

////////////////////////////////////////
//Return the size of the queue
////////////////////////////////////////
int priorityQueue::sizeOf(){
        return size;
}
```

```
////////////////////////////////////////
//Placing a node upon the queue
////////////////////////////////////////
void priorityQueue::push(node *element){

        bool complete = false;
        int position = 0;
        node *searchElement;
        CString test;

        if(size == 0){
           queue[0] = element;
                size++;
        }else{
                //the current element being searched is the first element in the queue
                searchElement = queue[0];

                //do while the item is incomplete
                do{
                        //if the heuristic is less than the current searchelements heuristic
                        if(element->getHeuristic() < searchElement->getHeuristic()){
                                move(position,element);
                                complete = true;
                                size++;
                        }else{
                                //search the next element in the queue
                                position++;

                                //if we have hit the last element in the queue
                                if(position == size){
                                   queue[position] = element;
                                        complete = true;
                                        size++;
                                }else{
                                        //set the search to the next element in the queue
                                   searchElement = queue[position];
                                }
                        }

                }while((!complete));
        }

}

////////////////////////////////////////
//Pop an item off the queue
////////////////////////////////////////
node* priorityQueue::pop(){
   node *returnNode,tempNode;
```

```
        int position = 0;

        returnNode = queue[position];

        do{
                queue[position] = queue[position+1];
           position++;
        }while(position <= size);

        size--;

        return returnNode;
}

//////////////////////////////////////////
//Moves all nodes up a position
//////////////////////////////////////////
void priorityQueue::move(int position, node *element){

        node *temp,*temp2;

        //repeat until we hit the end of the queue
        while(position <= size){
                //make a copy of the original data at the point
      temp = queue[position];
                //Insert the new data
                queue[position] = element;
                //move to the next position in the queue
                element = temp;
                position++;
        };
}

//////////////////////////////////////////////////
//Display the entire queue
//////////////////////////////////////////////////
void priorityQueue::display(){
   int counter = 0;
        CString test;

        if(size != 0){
                test.Format(_T("PRIORITY QUEUE\n==============\n"));
                while(counter!=size){
                        CString append;
                        append.Format(_T("%d.Pos = %d,%d : H =
%f\n"),counter,queue[counter]->getPosition().x,queue[counter]-
>getPosition().y,queue[counter]->getHeuristic());
                        counter++;
                        test.Append(append);
                };
                MessageBox(NULL,test,_T("Priority
queue"),MB_SETFOREGROUND);
        }
}

//////////////////////////////////////////////////
//Check to see if the queue is empty
//////////////////////////////////////////////////
bool priorityQueue::isEmpty(){

        if(size==0){
           return true;
        }else{
                return false;
        }

}

//////////////////////////////////////////////////
//Refresh the entire queue
//////////////////////////////////////////////////
void priorityQueue::rePosition(node *element){
   bool complete = false;
   int counter = 0;

        //we need to delete the original position of the node
        do{
                if(queue[counter] == element){
                        do{
                          queue[counter] = queue[counter+1];
                                counter++;
                        }while(counter < size);
                        complete = true;

                }
           counter++;
        }while(!complete);

        size--;

        priorityQueue::push(element);

}
```

Performancetracker.h

```
//////////////////////////////////////////////////////////////////////
// performanceTracker.h : interface of the performanceTracker class
//////////////////////////////////////////////////////////////////////

#pragma once
#include "stopWatch.h"

class performanceTracker
{
private:
        int paths,deadPaths,search;
        stopWatch timer;

public:
        performanceTracker(void);
        void addSearch();
        void addStep();
        void reset();
        void display(int x);
public:
        ~performanceTracker(void);
};
```

performanceTracker.cpp

```
////////////////////////////////////////////////////////////
// performanceTracker.cpp : implementation of the performanceTracker class
////////////////////////////////////////////////////////////

#include "StdAfx.h"
#include "performanceTracker.h"
#include "stopWatch.h"

performanceTracker::performanceTracker(void)
{
   paths = 0;
         search = 0;
}

performanceTracker::~performanceTracker(void)
{
}

void performanceTracker::addSearch(){
   search++;
}

void performanceTracker::addStep(){
   paths++;
}


void performanceTracker::display(int x){

         double time;
         time = timer.finish(paths);

         CString test;

         test.Format(_T("ALGORITHM
PERFORMANCE\n=================\nTotal searches : %d \nTotal paths created
: %d \nTotal dead paths: %d \nTotal Paths on found route: %d\nTotal time :
%f"),search,paths,paths - x,x,time);
         MessageBox(NULL,test,_T("RESULT OF
SEARCH"),MB_SETFOREGROUND);
}
```

stopwatch.h

```
///////////////////////////////////////////////////
// stopWatch.h : interface of the stopWatch class
///////////////////////////////////////////////////

#pragma once

class stopWatch
{
private:
	clock_t go, stop;
public:
	stopWatch(void);
	double finish(int paths);
public:
	~stopWatch(void);
};
```

stopwatch.cpp

```
//////////////////////////////////////////////////////////////////////
// stopWatch.cpp : implementation of the stopWatch class
//////////////////////////////////////////////////////////////////////

#include "StdAfx.h"
#include "stopWatch.h"
#include <ctime>

stopWatch::stopWatch(void)
{
	CString test;
	go = clock();
}

stopWatch::~stopWatch(void)
{
}

double stopWatch::finish(int paths){

	double time;

	stop = clock();
   time = (double)(stop - go) /CLOCKS_PER_SEC;

	//time -=(paths*.2);

	return time;
}
```

Stdafx.h

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently,
// but are changed infrequently

#pragma once

#ifndef _SECURE_ATL
#define _SECURE_ATL 1
#endif

#ifndef VC_EXTRALEAN
#define VC_EXTRALEAN            // Exclude rarely-used stuff from Windows
headers
#endif

// Modify the following defines if you have to target a platform prior to the ones
specified below.
// Refer to MSDN for the latest info on corresponding values for different platforms.
#ifndef WINVER                          // Allow use of features specific to
Windows XP or later.
#define WINVER 0x0501           // Change this to the appropriate value to target
other versions of Windows.
#endif

#ifndef _WIN32_WINNT            // Allow use of features specific to Windows XP
or later.
#define _WIN32_WINNT 0x0501     // Change this to the appropriate value to target
other versions of Windows.
#endif

#ifndef _WIN32_WINDOWS                  // Allow use of features specific to
Windows 98 or later.
#define _WIN32_WINDOWS 0x0410 // Change this to the appropriate value to target
Windows Me or later.
#endif

#ifndef _WIN32_IE               // Allow use of features specific to IE 6.0 or later.
#define _WIN32_IE 0x0600  // Change this to the appropriate value to target other
versions of IE.
#endif

#define _ATL_CSTRING_EXPLICIT_CONSTRUCTORS // some CString constructors
will be explicit

// turns off MFC's hiding of some common and often safely ignored warning messages
#define _AFX_ALL_WARNINGS
```

```
#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions

#include <afxdisp.h>        // MFC Automation classes

#ifndef _AFX_NO_OLE_SUPPORT
#include <afxdtctl.h>           // MFC support for Internet Explorer 4 Common Controls
#endif
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>                     // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT

#ifdef _UNICODE
#if defined _M_IX86
#pragma comment(linker,"/manifestdependency:\"type='win32'
name='Microsoft.Windows.Common-Controls' version='6.0.0.0'
processorArchitecture='x86' publicKeyToken='6595b64144ccf1df' language='*'\"")
#elif defined _M_IA64
#pragma comment(linker,"/manifestdependency:\"type='win32'
name='Microsoft.Windows.Common-Controls' version='6.0.0.0'
processorArchitecture='ia64' publicKeyToken='6595b64144ccf1df' language='*'\"")
#elif defined _M_X64
#pragma comment(linker,"/manifestdependency:\"type='win32'
name='Microsoft.Windows.Common-Controls' version='6.0.0.0'
processorArchitecture='amd64' publicKeyToken='6595b64144ccf1df' language='*'\"")
#else
#pragma comment(linker,"/manifestdependency:\"type='win32'
name='Microsoft.Windows.Common-Controls' version='6.0.0.0'
processorArchitecture='*' publicKeyToken='6595b64144ccf1df' language='*'\"")
#endif
#endif
```

Stdafx.cpp

```
// stdafx.cpp : source file that includes just the standard includes
// pathfinder.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```