Carleton University

COMP 4905 Honors Project

Testing an Analysis of D* Algorithms

By: Zoltan Moori

Advisor: Michel Barbeau, Computer Science

April 16th, 2008

Abstract

Path finding is one of the fundamental aspects of most game development projects. Path finding algorithms are used often and hence need to be efficient, while maintaining realistic movements of the non-player characters (NPCs). In the past, developers had resorted to cheating [Bough et al., 2004] [Scott, 2002] to improve performance, where information that would normally be unavailable to a player character (PC) is used to help the NPCs in determining their path. In recent years, the drastic improvements in video cards and their drivers freed up CPU cycles, which can be devoted to other functions like AI algorithms and path finding.

One of the most widely used graph search algorithms in game development is the A* algorithm [Hart et al., 1968]. A* is an efficient algorithm which computes the shortest path to a desired node in a graph, taking into account various obstacles while minimizing the cost. While it is efficient, it does have its drawbacks. The most significant drawback of the algorithm is that if the world includes dynamic obstacles, it may become fairly inefficient to use due to the fact that that the entire path needs to be calculated from scratch before the entity is able to make a move. Every game tick would have to check if the calculated path became obstructed, and if it has, the remaining path segment would have to be recalculated.

There are several variations based on the A* algorithm, which are designed to deal with constraints imposed by dynamic environments. The purpose of this project is to examine some of these approaches, and implement two of them to test its performance against each other and the static A* algorithm.

Acknowledgements

This project would not have been possible without the contributions of the following people:

My supervising professor, Dr Michel Barbeau, who provided me with a functional game engine to help me through the initial hurdles, and whose suggestions and support have helped me when the situation looked the bleakest, I would like to say, thank you.

My brother, Thomas Moori, who spent considerable amount of his free time proofreading this paper, while his own schedule was almost as pressing as mine.

I would also like to thank all the people who contributed or were involved in the publication of AI Game Programming Wisdom. Without the insightful articles contained in the book, I would have had a much more challenging semester.

Table of Contents

1. Introduction
   - 1.1 Artificial Intelligence in game development 7
   - 1.2 Informal Definition and Purpose of AI 7
   - 1.3 Challenges in Game AI 8
   - 1.4 Position of Game AI's in the Development Cycle 9
   - 1.5 Right Tool for the Right Job 12
   - 1.6 Future of AI 12

2. Map representation
   - 2.1 Map Partitioning 13
   - 2.2 Graph Search 17

3 A* search
   - 3.1 The Algorithm 22
   - 3.2 Heuristics 25
     - 3.2.1 Effect of the Heuristic Value on A* 26
     - 3.2.2 Calculating Distance to Goal Heuristic 26
     - 3.2.3 Breaking Ties 27
   - 3.3 Analysis of A* 28
   - 3.4 Variations of A* for static environments 31
   - 3.5 Pathfinding for Multiple Agents 32

4. Dynamic Pathfinding
   - 4.1 Search Direction 33
   - 4.2 Approaches to deal with map changes 35
   - 4.3 Pathfinding versus navigation 38
   - 4.4 A* Variants for Dynamic Navigation
     - 4.4.1 D* Lite 39
     - 4.4.2 D* and Focused D* 41
       - 4.4.2.1 Initialization 42
       - 4.4.2.2 PROCESS-STATE 43
       - 4.4.2.3 Helper functions 44
       - 4.4.2.3 D* and FD* Node Expansion 44

5. Implementation Details
   - 5.1 Map Design Decisions 46
   - 5.2 Non-Player Characters 49
   - 5.3 User Interface 51
   - 5.4 Pathfinding / Navigation Engines 53
   - 5.5 Relevant Variables for Testing 55
   - 5.6 Adding New Pathfinders 55

6. The Experiment
   - 6.1 Experimental Method and Justification 56
   - 6.2 Experimental Results 58
   - 6.3 Observations 58

7. Conclusion
   - 7.1 Summary 59
   - 7.2 Future Improvements 60

List of Figures

FIGURE 1: Concave polygons 16
FIGURE 2: Navigation mesh 16
FIGURE 3: Points of visibility 17
FIGURE 4: Dijkstra algorithm on a grid map 19
FIGURE 5: Wall following cycle 20
FIGURE 6: Best-first search on a grid map 21
FIGURE 7: A* search result on an open terrain 24
FIGURE 8: A* search result on a maze map (forward A* search) 24
FIGURE 9: Breaking ties 27
FIGURE 10: Reverse A* search tree 34
FIGURE 11: Influence map 37
FIGURE 12: Monster behavior 49
FIGURE 13: Statistics window 52
FIGURE 14: Message sequence chart of main actors 54

List of Tables

TABLE 1: A* operations 30
TABLE 2: D* Lite and A* node state comparison 40
TALBE 3: Characters used for map creation 48
TABLE 4: Game key shortcuts 51
TABLE 5: world2 test results 58
TABLE 6: world3 test results 58

1. Introduction

1.1 Artificial Intelligence in game development

Games are developed to entertain people. Much of the entertainment value of a game comes from a combination of numerous factors, such as graphics, level design, character development (for role playing games), as well as various interactions with computer controlled entities. While graphics, sound effects and design of the levels are important, since they provide the majority of the initial "WOW factor", but once the player completes the game for the first time, the graphics and levels will generally stay the same for the next game. Unpredictability provided by the NPCs ensures that a player playing the game the second time around encounters different situations than they did the first time, thereby adding significant replay value to the product. Artificial Intelligence methods are responsible for controlling in game entities (including allied and opposing characters), the story line, and weather patterns.

1.2 Informal Definition and Purpose of AI

The term Artificial Intelligence is somewhat misleading as it applies to the Gaming Industry, since its meaning varies from its traditional academic counterpart. The field of Artificial Intelligence in Computer Science implies the use of non-deterministic algorithms, which modify their internal structure in one way or another in light of past experiences. Neural networks change the weights of their connections; machine learning algorithms create new rules; and genetic algorithms create a new and different

population by combining genes of the more successful individuals (based on a fitness function) from the previous generation. All of these changes to the internal workings of the algorithm imply that the same input at different stages of the algorithm's evolution will result in different outputs. This, however, is not necessarily the case for Artificial Intelligence algorithms used in the Gaming Industry. While traditional AI algorithms are certainly used for game development, the term Artificial Intelligence in the Gaming Industry refers to any algorithm which makes entities of the game seem intelligent. As such, finite state machines and path finding algorithms are also included in this category. To alleviate some confusion, the term Strong AI is used for traditional AI methods, while the deterministic algorithms are often referred to as Weak AI algorithms [Bough et al., 2004]. From this point on in the report, Artificial Intelligence will be used as it pertains to game development.

1.3 Challenges in Game AI

The goal of Artificial Intelligence as mentioned before is to simulate behaviors from the game agents that are both believable and challenging. An opponent whose actions can be predicted most of the time does not provide much challenge or entertainment for a player. Likewise, if the opponent is completely unpredictable and uses all the information available in the game to beat the player (player's position, status; location of resources, doors etc), it may present the player with an impossible scenario, which very few view as rewarding. It is fairly easy to implement unbeatable opponents or completely clueless ones, the trick is to balance out the challenge and reward so that a

player can predict and counter some situations, where in others they are outmatched by the computer [Scott, 2002].

Due to the non-deterministic nature of Strong-AI algorithms, the debugging and testing process requires much more effort from the development team. Such algorithms often require additional time to train them properly, as well as strict guidelines for the parameters, so when playing against a bad player they don't learn incorrect lessons. Doing so may render their future decisions against skilled players trivial, or result in unexpected and unrealistic behavior from the agents. The deterministic algorithms on the other hand are fairly straight forward to understand, implement, debug and test. On many occasions Implementing AI in games has been a rush job prior to release [Tozour, 2002], especially during the era when most of the focus was on graphics. Developers opted for the tried and true methods for AI, such as using Finite State Machines to control the behavior of the agents as opposed to experimenting with Strong-AI methods like A-life, which would have taken significantly more time to implement. While some Strong-AI methods have had reasonable success (Black & White by Lionhead Studios, or Sims series by Maxis), most developers still feel that the effort required in implementing, fine tuning, testing and debugging outweighs the benefits of such technologies, and hence are seldom used.

1.4 Position of Game AI's in the Development Cycle

In the last decade, the majority of the research focus has been on improving graphics performance, since it was the most CPU intensive operation performed by games, and

depriving CPU cycles from other elements like physics engine, or Artificial Intelligence. With the significant improvements to graphics technology over the years, both on the software and hardware level, more CPU cycles were available for other functions that were previously neglected, and were now allowed to thrive. It would be a logical expectation to see AI technologies skyrocket under such circumstances, and while they have received much more attention lately than in the past, no real groundbreaking advancements have been made.

Even in current game development, the AI subsystem is one of the last to get implemented, but for different reasons. Generally the decisions required from the AI are heavily tied to other aspects of the game. Path finding for instance requires level designers to have completed at least a rudimentary version of the maps, so the algorithm could be tested against it, and optimizations specific to the game can begin. Decision making for the NPCs largely depends on the actions available to the entity, which are usually tweaked until well into the development cycle to balance the reward versus challenge tradeoffs. Last subsystem to be implemented however does not mean the work on the AI does not start until the end of the development cycle. In recent projects, teams are designated exclusively for developing AI at the design stage [Tozour, 2002]. This enables the team to start working on the general model of the AI, which algorithms will be used, and how they will interact with each other. Some of the independent components, like weather system, can also be implemented. That being said, even if AI could be the very first subsystem implemented, with all the parameters required for it decided during design stage, it would still not happen. The reason is fairly simple: marketing. A game needs a fan base before the product hits the shelves. The

marketing team needs screenshots, and short video captures to be able to showcase the game. Without graphics and level design, it is very hard to impress potential fans with stick figures moving around in a bland world even if they do so with very realistic, human-like behavior. The AI portion of the game is usually mentioned in interviews, at which point the developers should have a really good idea of how the system is going to work, but the implementation does not need to be complete.

A discussion of game AI is not complete without mentioning cheating. Cheating is one of the most common AI strategy employed in games [Bough et al., 2004]. Some people are very opposed to this practice, and think that the computer should only use information which would be available to a player under the same circumstances. While it may seem unfair, and may give the impression that the computer has the advantage, in reality the AI is at a severe disadvantage. Other than the fact that the computer may be responsible for controlling multiple opponents in real-time simulation (RTS) games, it also has a pre-defined set of algorithms to work with (at least until learning algorithms become more manageable for the developers). Once the player recognizes these patterns, and is able to predict them, the computer will need all the help it can get to provide a challenging experience for the player. Most players prefer to have a cheating opponent provide a reasonable challenge, than an honest one unable to entertain them. This is yet another tradeoff that the developers have to balance carefully. Cheating, if used too often or in an obvious manner may indeed be detrimental to the player's experience, and should be avoided.

1.5 Right Tool for the Right Job

Different AI algorithms are best used for their specific problem domain. There is no one algorithm that will be able to control all the actions of every entity. It is important to examine what each entity is expected to perform and then select the best algorithm for that purpose [Tozour, 2002]. Most AI models for modern games include a mixture of different AI methodologies, each custom tailored for their purpose. While one model may work ideally for one game, it is not likely that they can be adapted to another game without major modifications. This is especially true for games of different genres. Players in different games have different goals and methods available to achieve those goals, and the same is true for their opponents. It is feasible that some developers reuse a significant portion of their previously released game for the new game they are developing. Most likely candidates for this would be the sport simulation games which come out every year, such as NHL, NBA or Madden Football line developed by EA Sports. However the main reason why such an approach is possible is because the game's internal mechanics remain similar from year to year.

1.6 Future of AI

There are a lot of opportunities for research in the AI area, including realistic speech generated by the computer able to portray emotion and the personality of the character [Isla eta al., 2002] (currently a lot of games use pre-recorded snippets to advance the story line), or possibly a developer-friendly unified API for some of the strong-AI technique implementations. Research in the Computer Science AI field is ongoing on a

wide array of topics; however there are some in the gaming industry who question their relevance for game development. Nareyek argues that most of the research done currently is very specialized and only applicable to a very narrow set of problems [Nareyek, 2004]. He also states that a lot of the resulting algorithms do not function well under real-time processing constraints, making them irrelevant for the gaming industry. The research done for robotics is much more applicable to game development (and vice versa), as the two fields have similar operating parameters, granted the method of input is very different. While robots have to use sensors to learn about the environment around them, an entity in a virtual world only has to query the world to obtain the relevant information, but the approach used to interpret the environment to aid in the decision making process is very similar.

2. Map representation

2.1 Map Partitioning

Before beginning discussion on the specifics of path finding, it is important to examine the maps used for games and how they influence the algorithms used. It should be mentioned at this point, that all path finding algorithms work with graphs as their data structure, meaning that any map needs to be converted to a graph in one way or another. Maps can be categorized based on different criteria. The first and most obvious one is related to the representation of the map.

A lot of the older games used a collection of uniform tiles to construct the levels, using a two dimensional array for storage, which can be accessed by row and column co-

ordinates to obtain various locations on the map. Most common shape for the tiles is square (since it is easy to translate into a text file or data structure for storage), but it is possible to use others like triangles and hexagons as well. The values stored for each location are generally custom structures containing all relevant information associated with that location. Such structures may contain terrain types, event scripts to tell an entity what action to perform [Orkin, 2002], links to resources (images, animations, sounds), or weights representing difficulty of travelling through that tile. 0 may stand for impassable walls, 1 may be used for clear terrain which is easy to traverse, and 2 may represent rougher terrains. While this method is used only in very few modern games, it is still valuable learning tool for those interested in eventually working in the industry. The tile environment simplifies most of the algorithms significantly. There is no vector algebra to deal with when determining different steering forces acting on an entity, either from the physics engine, or associated with other algorithms such as obstacle avoidance. Most of the non-AI related algorithms are also simplified, there is no object rendering, no clipping pane, and a physics engine may not even be required (with the exception of gravity if the game allows jumping or falling). There is a finite set of directions available to move from any given grid position, depending on the shape of the grids. Such maps are also are extremely easy to create and maintain, and no processing is required to convert it to a graph, since all adjacent tiles are assumed to be connected.

Modern games use three dimensional maps made up of a collection of textured polygons. Entity movements are described by vectors, which instead of the finite directions, as it was the case for the tiled environment, can point any direction the

character faces. Locations are described using floating point x, y, and possibly z co-ordinates (the z co-ordinate may be implied unless the map contains multiple floors or layers). One of the challenges with such maps from the AI perspective is converting the infinite positions located on the planes representing the world into a finite set of interconnected nodes for the path finding algorithms. This may be achieved in a few different ways, but all of them involve further processing of the map to break it down into more manageable format. The two most common methods used are Navigation Meshes and Points of Visibility.

- Navigation Mesh is an overlay created by breaking down the unobstructed areas of the map into convex polygons. Generating a mesh for a map is in essence converting a continuous world into a tile based map, with a few exceptions. The tiles do not need to be square, or have the same size or shape, but they do need to be convex. Convex polygons are required so when traveling from any point of a polygon to an adjacent polygon it will not cross through a third polygon, as illustrated by FIGURE 1. The navigation meshes can be automatically generated by various algorithms depending on the shape and style of meshing desired; however they should be visually inspected by level designers especially when used on a map with varying elevations. FIGURE 2 shows an example of a NAV mesh [Vincent, 2007].

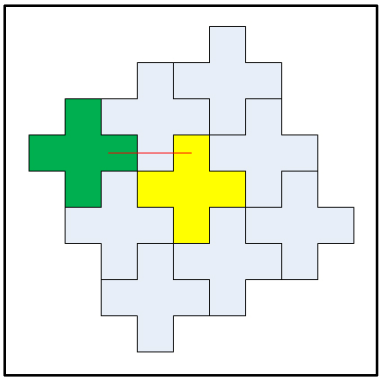

FIGURE 1: Traversing between adjacent polygons may cross a third polygon as indicated by the red line

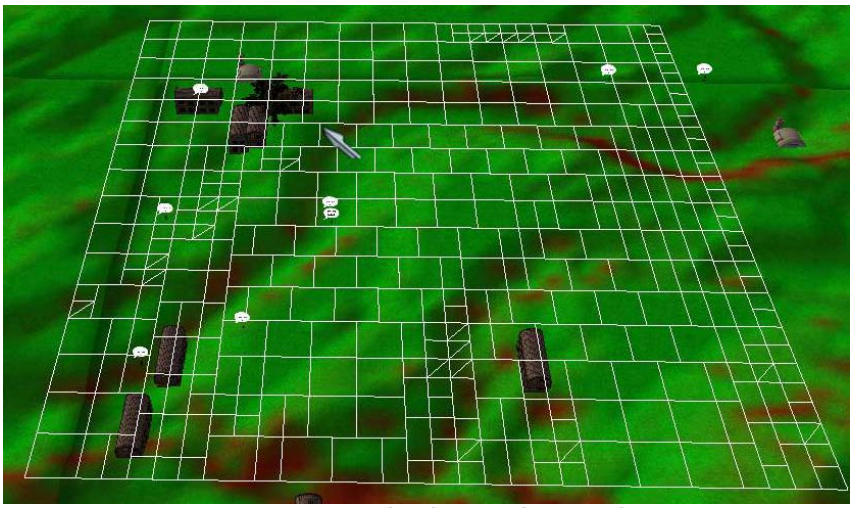

FIGURE 2: Navigation mesh example

- The point of visibility method involves placing a set of interconnected waypoints across the map along the routes where the player can travel. FIGURE 3 shows an example of PoV placement in a maze [Bough et al., 2004]. This method is analogous to drawing a graph over the map including the points of interests where neighbouring waypoints can be reached by a straight line. The waypoints can be placed manually, or there are also automation tools are available for this purpose. One important aspect to keep in mind when selecting positions for the waypoints is that every position a player can reach needs to be in line of sight of at least one waypoint, otherwise the game entities will not be able to reach that

spot. One issue with using waypoints is the jagged movement when an agent reaches one waypoint and takes a sharp turn towards the next one creating an unnatural looking path. Methods like spline interpolation [Coleman] help correct this issue.

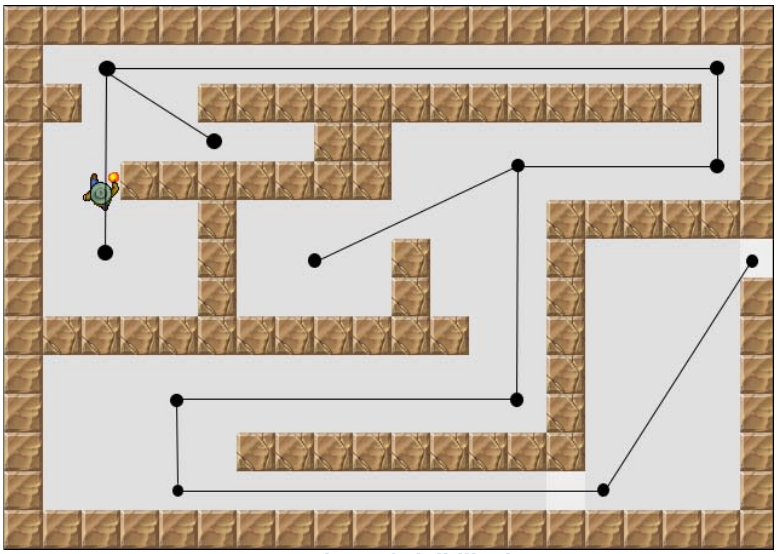

FIGURE 3: Points of visibility in a maze

Like it was the case with Navigation Meshes, the structure used for path management is usually stored in a file separate from the map, and needs to be constructed before the product ships. Since the structures are pre-processed, they will not be able to handle dynamic obstacles well, as the structure, which ships with the product, would have to be updated every time a path becomes blocked. This problem is left to the pathfinding algorithm to resolve.

2.2 Graph Search

A graph consists of a set of nodes connected by edges. The goal of the pathfinding algorithm is to find a suitable path from one node to another. The term suitable is used,

since the shortest path is not always the best path, as there may be many other factors to consider when selecting a route. Such factors may include travel time, scenery along the route, presence of hostile entities, or difficulty of the terrain. During the search algorithm, nodes can go through three different states. Initially they are undiscovered. When the algorithm reaches them, they become discovered, and when all the neighbours of a node's have been examined, the node becomes explored. The discovered nodes are often referred to as frontier, while the combination of the explored and discovered nodes form a flood generated by the algorithm, which represents all the nodes that were examined. The flood of the algorithm gives a good indication of how much work was involved in finding that particular path.

There are two different approaches to finding a path: Breadth-first and Depth-first search. Breath-first search starts from the initial position and recursively expands outward in every direction like a ripple on a pond, while keeping track of where each node was discovered from. As the algorithm expands the frontier, it will eventually find the goal node, and backtracking from that node is able to construct the path. Dijkstra's algorithm [Dijkstra, 1959] uses such an approach and guarantees to return a path of the shortest length if one exists. If more than one such path exists it will return the one it finds first. As illustrated on FIGURE 4, the amount of nodes checked is fairly large, and if an obstacle is introduced along the path, it significantly increases the examined grids in every direction.

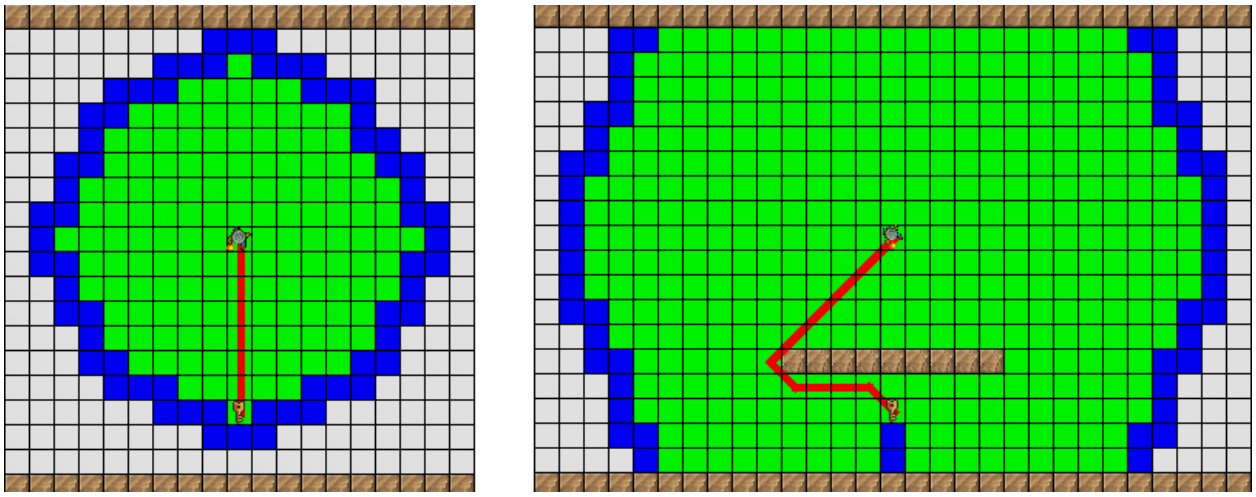

FIGURE 4: Dijkstra algorithm on a grid map, green tiles indicate explored nodes, blue tiles are the discovered nodes

Depth-first search on the other hand, takes the opposite approach. The algorithm picks one direction or path to explore, and sticks to that direction, only backtracking when no other option is available. The follow right (or left) wall strategy is very similar in nature and is used by many players when exploring a dungeon or even a more open terrain. The cycle for the follow the wall method can be pre-computed, and does not need to be done in real time, and it's just the matter of connecting the starting and ending point to that cycle to complete the path (FIGURE 5). This method may not produce the shortest path, and it may not work well in for graphs with a lot of cycles.

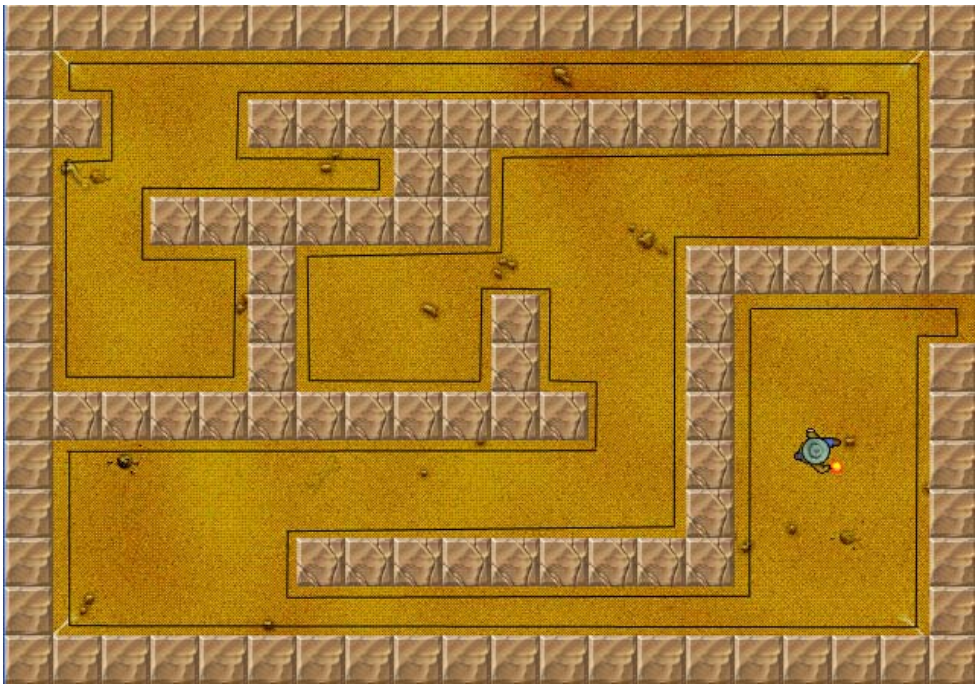

FIGURE 5: Wall following cycle created for a static map which can be traversed in either direction

None of the methods mentioned above use any knowledge of where the goal lies until the actual goal node is discovered. In order to guide the algorithm toward the goal more purposefully and to eliminate discovering useless nodes (which are unlikely to be part of the solution), information about the direction or the distance to the goal can be used to improve the responsiveness or efficiency of the search. Since the exact path to the goal is not known, the distance to the goal can only be estimated. Such estimates are referred to as heuristic values. The best-first search is an algorithm that uses distance to goal as a heuristic. As the name implies, it is a greedy depth-first search algorithm that will selects nodes which will bring it closer to its goal. As long as that distance keeps decreasing with each step, it will stick to that path. The algorithm works fairly well in an open terrain with few obstacles, but may produce unnatural looking and far from optimal path otherwise, as can be seen on FIGURE 6 [Anon, 2003]. Like other

depth-first search methods, best-first is not guaranteed to find an optimal path, however in the best case, it will find a reasonable path with significantly less resources than the Dijkstra's algorithm would; while in the worst case it will need to examine almost as many nodes, and come up with a path that is unacceptable.

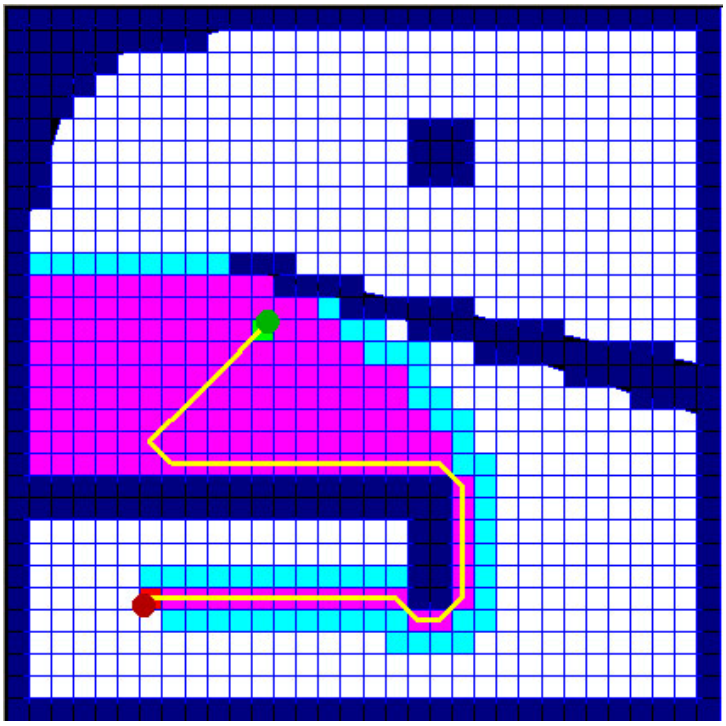

FIGURE 6: Best-first search on a grid map

No matter what search algorithm is used, the larger the graph, the longer the search will take. In order to improve the performance, nodes can be grouped together to form higher level nodes, from which a higher level graph with considerably less nodes can be constructed [Botea et al, 2004]. For example, a node can be used to represent a room, and grouping the nodes for one floor a new node representing the whole floor is created. Nodes representing floors can be further combined to form a higher level node for the building, and so on. This hierarchical structure allows higher level graphs to be

processed first, forming a rough idea of the final path, and then starting to process the lower level graphs close to the initial point to create subpath. With first subpath constructed, the entity can start moving toward the goal while the rest of the path segments are still being computed.

While finding a path from point A to B is the primary objective of the pathfinder, and is the focus of this paper, it is worth mentioning that such algorithms can be used for other purposes as well. Repeatedly finding paths from random start and goal locations can help detect choke points on the map, which can be used either to modify the terrain, or to help the agent establish ambush spots. By selecting a certain terrain type as a start point, and an unreachable goal, while only allowing the algorithm to explore certain types of terrains can result in flood fills useful for finding forests, water masses, or interconnected road paths [Higgins, 2002a]. Pathfinding can also be used to mimic exploratory behavior, and conducting traffic analysis just to mention a few.

3 A* search

3.1 The Algorithm

Combining the breadth-first nature of the Dijkstra's algorithm with the heuristic driven best-first method forms the basis for the A* algorithm. The approach to handling the nodes is essentially identical to Dijkstra's algorithm, and an open list is used to keep track of the frontier nodes, while a closed list maintains the discovered nodes. The main difference is the way weights are calculated, resulting in the frontier not extending equally in all directions, but more toward the goal and less in other directions. The other

directions need to be examined in case the promising direction turns out to be a dead end or a detour. The weights associated with the edges are a combination of a heuristic value, the distance travelled so far to that node, and possibly other values which can be used to modify the behavior of the path finder to deal with dynamic objects or other factors. The dynamic modifiers will be discussed in more detail later in the report. Each node keeps track of the following information: g for the distance for the most efficient path found to this node, h for the heuristic value, f is the estimated path length to the goal through that node, which in the base A* algorithm is the sum of g and h, and finally the parent of the node, that is used to be able to backtrack to construct the path once the goal is reached. It is important to note that the cost of travel between two nodes has to be greater than zero in order to avoid infinite loops.

As mentioned before, if the A* algorithm encounters an obstacle along the best estimated path, it will expand its search space. The same experiment as shown on FIGURE 4 when executed using the A* algorithm yields a much smaller search space, as illustrated on FIGURE 7. The red line represents the path to the goal, and the pink and yellow lines are the heuristic paths of some of the closed nodes. The first segment of the path estimates represents the g value, while the second segment (crossing through the obstacle) is the h value. All the estimated path lengths of the closed nodes is less than or equal to the shortest path length found upon completion. The longer the detour the agent is forced to make, the bigger the flood generated by the algorithm will be. On maze like maps with lot of obstacles and deviations from the straight path towards the goal, the algorithm is forced to examine nearly the same amount of nodes

as Dijkstra's algorithm did, as illustrated on FIGURE 8. The search tree and its direction will be discussed in a later section.

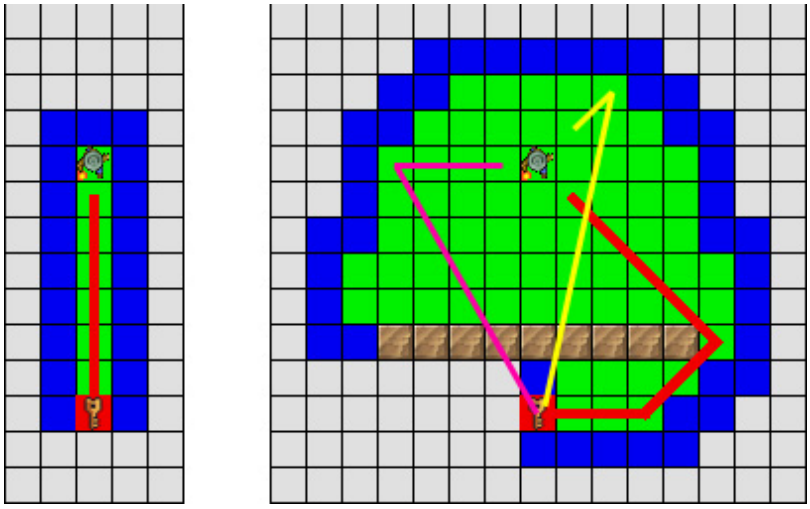

FIGURE 7: A* search result on an open terrain

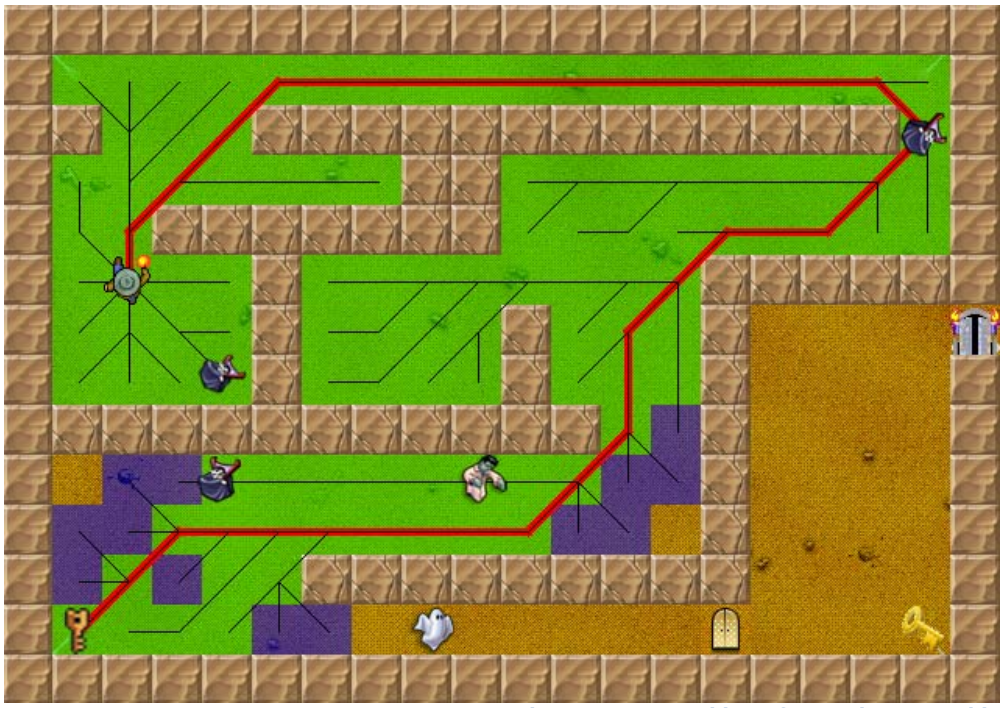

FIGURE 8: A* search result on a maze map (search tree generated by a forward A* search)

The basic pseudo code for the A* algorithm is as follows is listed in APPENDIX A. In each iteration, the node in the open list (which only contains the start node initially) with the lowest f value is selected, and all of its neighbours are checked. If the neighbour node has not been examined in the past, its g, h, f and parent values are populated to reflect where it has been discovered from. If the neighbour is either in the open or closed list, it means it has been previously examined, and the previous f value is compared to the new f value, and if a more efficient path for the neighbour has been found, its g, f and parent values get updated. This change may affect its successor nodes as well. There are two approaches to update the graph to keep it consistent. The passive approach is to reinsert the node back into the open list, and if its weight is low enough it will eventually be examined once again, resulting in its successors to be updated too, if needed. The active update consists of propagating the changes as they occur and placing the node into the closed list. Both methods will yield an optimal result, but active updates require more processing time, but less storage space. When all the neighbours of the best fit node are discovered, it is placed on the closed list and the same process is performed on the next node in the open list with the lowest f value. The algorithm terminates when the goal node is discovered, or the open list runs out of nodes, signifying no path is available.

3.2 Heuristics

The heuristic value used has a large impact on how the A* algorithm will perform. The method used to estimate distance to goal depends on the type of map used, and desired behavior of the agent.

3.2.1 Effect of the Heuristic Value on A*

If the heuristic value underestimates the distance, an optimal path will still be found if one exists. However more nodes will be explored in this case, since each step towards the goal will not seem to get the entity as close to the goal as the heuristic indicated. Nodes in other direction will be explored further, since steps away from the goal don't look as bad of a choice as they are in reality, and the algorithm will become less goal-oriented. In the extreme case when the estimate is always 0, the A* algorithm deteriorate into Dijkstra's algorithm [Patel, 2000].

If the heuristic value is overestimated, each step away from the goal will have a magnified effect making it seem like the that direction is not worth exploring, hence the algorithm will avoid those steps, and focuses more on options which lie directly towards the path. In this case an optimal path is no longer guaranteed, since the algorithm can get stuck in a local minimum, and in the extreme case the algorithm will perform similar to best-first search.

In order to obtain an efficient shortest path, the heuristics should try to determine the distance to goal as accurately as possible without overestimating it. If the heuristic never overestimates the cost required to reach the goal it is referred to as admissible.

3.2.2 Calculating Distance to Goal Heuristic

In a continuous map environment, the calculation of the heuristic value boils down to applying Pythagorean Theorem with the difference in x and y locations to obtain:

$h = \sqrt{\Delta x^2 + \Delta y^2}$.

If the movement is restricted to the 4 cardinal directions on a grid map, the Manhattan distance is used: $h = \Delta x + \Delta y$.

A grid map that allows diagonal movement, diagonal movements may count as 1 movement, or $\sqrt{2}$ depending on implementation. Let d represent the diagonal movement cost, the estimate works out to be: $h = d * \min(\Delta x, \Delta y) + |\Delta x - \Delta y|$. The minimum term is the shorter axis of the path, which is the number of times diagonal movements are necessary for the optimal path, while the rest will be traversed along the longer axis, as illustrated in FIGURE 9.

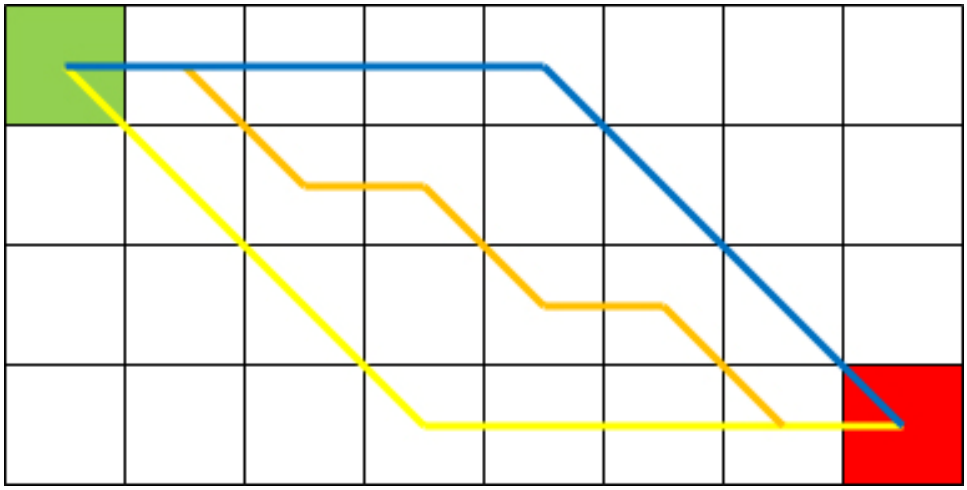

FIGURE 9: Multiple optimal paths from start to goal may be present

3.2.3 Breaking Ties

In some cases, multiple optimal paths may be present, as is the case in FIGURE 7, some more visually pleasing than others. Which path is returned by the search algorithm? It depends on how equally promising nodes (nodes with the same distance

from start and heuristic values) are treated when the algorithm asks for the best fit node from the Open List. This problem does not occur very often in continuous maps, where the locations are represented by floating point values, but is very common in tile based environment. If the fitness value of a set of nodes is the same, their heuristic value can be used as a tie breaker, otherwise there are several approaches possible to selecting from identically promising nodes [Patel, 2000].

- The first and easiest method is to not deal with it at all, and arrange nodes with equal fitness in the order they are inserted into the list. Due to the fact that neighbouring nodes are inserted in the same order each time, one direction will be favored over another resulting in the blue or yellow path in FIGURE 9.
- Another approach is to select one of the nodes with the same fitness at random. While this method is by no means perfect, it does tend to produce more visually pleasing paths.
- Other methods, as the one described in [Coleman, 2007] perform penalize undesired characteristics emerging while the path is computed by increasing the heuristic value, and use a second pass once the goal is reached to further refine the path.

3.3 Analysis of A*

In order to analyze the performance of this algorithm, and to decide what data structures to use, each operation needs to be examined and determined how often they will be executed. There are 3 different operations performed on the open list, namely;

remove best fit node, *insert node*, and *checking the list to see if a node is already present*. The closed list uses the same operations as the open list does, with the exception that it does not need to remove the best fit node, instead it removes a specific node to which a shorter path from goal was discovered than the one previously recorded. Luckily this situation does not arise very often [Higgins, 2002b], and for the purposes of the runtime analysis will be mostly ignored.

Each node examined is inserted into the open list at least once, and may or may not be removed from that list. Nodes may or may not be inserted into the closed list, and could be inserted more times if path upgrades are found. They can also get removed at most as many times as they have been added to the closed list. In order to analyze the algorithm, it is assumed that path upgrades for each node will be performed O(1) times. Accessing a node from either list, either for the purposes of updating the node's values if a faster path is found, or just to check if the list already contains that node may be called multiple times for any node. Every other operation in the algorithm is performed in linear time.

In summary, when the algorithm finishes, every discovered node was checked in the open list (and closed list) for existence, and then inserted into the open list only once. When discovered nodes become explored, it gets removed from open list, and inserted into closed list. Each of these function calls are performed once. The consists function is accessed O(neighbours(node)) times, once for each neighbour (assuming no path updates took place for its adjacent nodes). Accessing any node in either list should be efficient, as it will be called the most. TABLE 1 summarizes these results, x stands for

the number of explored nodes, d is the number of discovered nodes, e is for the number of edges tested during the algorithm, |N| and |E| are the number of nodes and edges in the connected portion of the subgraph respectively.

	Average case	Worst case
OpenListInsert	x + d	|N|
OpenListRemove	X	|N|
ClosedListInsert	X	|N|
AccessNode	e	2|E|

TABLE 1: Number of operations performed by A*

The most obvious choice for both lists would be a sorted list, so binary search can be used to check for or access any node in logarithmic time. Finding the lowest cost node would be constant time, since it's either the first or the last element of the list, depending on implementation, however if the list requires shifting of each element over after removal, the method degrades to linear time. Insertion would require logarithmic time to find the insertion point, and then depending on implementation constant or linear time for the actual insertion. The ideal structures to use depend on a variety of other factors as well. The number of agents using the pathfinder, the denseness of the graph, and the number of adjacent nodes for an average node all have consequences on the ratio of discovered and explored nodes. Other potential structures which could be used are outlined in [Patel, 2000] and [Higgins, 2002b]. A* has been proven to always return a shortest path if one exists, while examining at most as many nodes as any other algorithm finding the shortest path using the same heuristic [Dechter et al., 1985].

3.4 Variations of A* for static environments

There are several variations based on the A* algorithm to best suit problems of different domain, and only a few of them will be mentioned, as they are not the focus of this report.

PHA* [Felner et al., 2004] is used in the field of robotics, where a robot moves around in an unknown or partially unknown environment, and effort required to check the fitness of an unexplored node is not necessarily linear. The agent has to actually travel to the spot, and examine the surrounding nodes from that location. The order in which the nodes in A* are expanded would result in the robot having to travel back and forth from one side of the frontier to the other. As mentioned before, any algorithm finding the shortest path has to expand at least as many nodes as A* does. The additional challenge presented by this scenario is visiting all the required nodes while minimizing the distance travelled. While expanding a node the algorithm considers if it is worth taking a slight detour along the way to explore another fit node in the open list, or if it's worth travelling across the search space for the best fit node, or explore the next two nodes on the open list first which are much closer to the robot's current location.

IDA*[Korf, 1985] is used for large maps, or on devices with limited storage capacity. In order to limit the number of nodes on the open list, each iteration sets an upper bound on the value of f of the nodes to be expanded. Subsequent iterations relax the upper bound to get a broader view of the graph, achieving linear storage, as opposed to the exponential required for the A* algorithm.

HPA*[Botea et al, 2004] utilizes hierarchical partitioning of the map mentioned earlier. The map is divided into equal sized portions, referred to as clusters, and a limited set of portals are defined for each cluster indicating the positions where the agent can exit the current cluster. The determining of portals is done as a pre-processing step, which is followed by finding the shortest paths between each portal within a cluster, and then building a connection map between the clusters. When the agent requests a path between two points; the pathfinding problem boils down to finding a path between the start and goal position to their closest portal within their clusters, since every other path segment in between is pre-computed. This method does not yield an optimal path, but in a lot of cases it is close enough, especially when considering the amount of computation time saved. HPA* also works to some degree with dynamic maps, if a pre-computed path within a cluster is obstructed between two portals, A* can be used to determine a shortest path. As long as the portals do not get obstructed often, it handles dynamic objects fairly well. When a portal gets obstructed, the paths between the portals in the cluster have to be recomputed, and the connection map distances need to be updated as well.

3.5 Pathfinding for Multiple Agents

So far the discussion was mainly focused on moving an individual unit. Games where considerable amount of entities use the pathfinder can also use various optimization techniques described in [Higgins, 2002a], [Higgins, 2002b] and [Cain, 2002], dealing with a wide variety of topic from time shared pathfinding, calculating quick paths and splice paths for instantaneous reaction to a move command. When dealing with groups of entities moving towards a similar goal, instead of each unit performing path finding,

they can randomly assign a leader amongst themselves, and either follow that leader's path, which looks somewhat unnatural to have a group of entities approaching single file, or use other methods, like flocking described in [Reynolds, 1999]. The topic is out of the scope of this project, as the focus is more on single entity pathfinding.

4. Dynamic Pathfinding

As mentioned earlier, the maps generally pre-processed before shipping to create the appropriate structures to aid with path finding, and hence do not aid the algorithm with dynamic obstacles or changing terrain costs. Such conditions arise very frequently in modern games, and none of the algorithms mentioned up to this point deal with such constraints. If and when a dynamic obstacle is encountered, a new search would have to be started from the current location discarding all previously calculated information which may still be relevant. The highly dynamic games of today demand a better approach.

4.1 Search Direction

For the purposes of the static algorithm it makes little difference which direction, the search is conducted, both directions will yield an optimal result. The search tree generated in the two instances will vastly differ though as indicated on FIGURES 8 and 10. The tree limbs consist of a lot of straight paths, which indicate no tie breaking condition in use, and one direction is clearly preferred over another. As the agent moves along the path generated by the forward search, the tree generated will not be up to date for positions behind the player. It may not present a problem until the agent

encounters an obstacle along the path, in which case the whole tree would need to be adjusted to reflect the agent veering off the path.

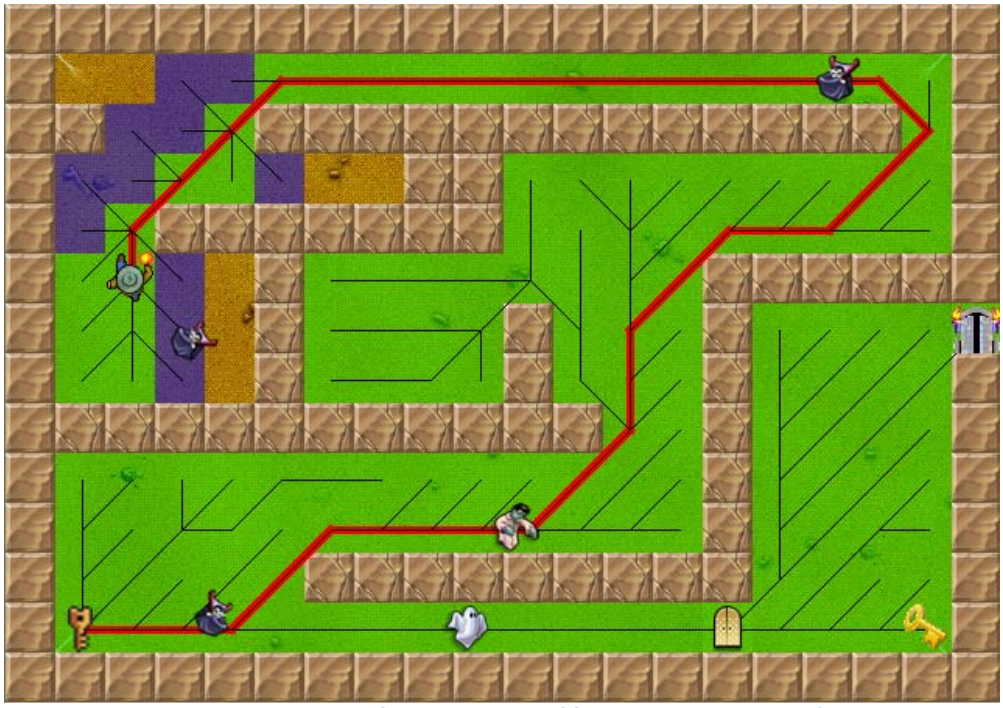

FIGURE 10: Search tree generated by a reverse A* search

The search tree stays consistent for the reverse search as the agent moves along its path. The tree still needs to be adjusted, however the effort required is significantly less. Once the first goal is reached, the initial path may also be available without any additional computation (the path from the key to the door on the bottom row tile of FIGURE 10). For these reasons, reverse search is usually used in dynamic environments.

No matter which method is used, great care has to be taken not to create cycles in the search tree while updating the map to reflect the changes occurring in the terrain.

4.2 Approaches to deal with map changes

The first step in facilitating dynamic pathfinding is to identify the dynamic elements, whether they are opposing units presenting threat to the agent, or temporary terrain condition changes due to congestion or weather.

The field of robotics has done extensive research on traversing through dynamic worlds. The problem domains in the two fields are very similar, and the algorithms developed to aid a robot reach their destinations can be adopted to work for a game environment. There are some differences in the restrictions imposed by the two problems. Both require real-time responses from the agent, but while it may be acceptable for a robot to stop for a short time to update its path after recognizing a discrepancy between the internal map and its sensors, such a reaction is very undesirable in a game situation. When a player tries to move units somewhere on a map, or the units encounter unexpected obstacles along the route, they are expected to deal with it seamlessly.

There is also a difference in how the agent acquires the information. Aside from their internal map representation (which at times may be partially or completely unknown), they rely on their sensors to provide information about their surroundings. As such, the range of its updating capabilities is very limited. An entity in a game could have any information about the map, including dynamic aspects as they happen. In this sense the game developers definitely have a huge advantage, however while robotics algorithms' main focus is on efficiency and correctness, game developers have to design a game that is fun and challenging for the player. Too much information can be

detrimental for a game, and part of the challenge becomes deciding how much of this information to utilize.

A popular way to deal with obstacle avoidance in the field of robotics is with the use of potential functions [Goodrich, 2002]. This method works fairly well in continuous environments, where movement is defined by a vector, but scales poorly to tile based games due to the difficulty to implement vector algebra properly with the a limited amount of possible directions utilized. Each dynamic object emits a concentric repelling or attracting force. When the entity enters the range of the force, it will modify its travel vector according to the nature and strength of the force. These forces are always active and do not need to be triggered, the only requirement is for the agent to recognize that it has dynamic force(s) acting on it. The pathfinding algorithm does not need to be modified as it only provides the agent with a general direction toward the goal, which will be altered along the way by these forces. While this method works well for dynamic obstacle avoidance, it does not handle terrain changes, and may even be detrimental in some cases. For instance, if a group of allied forces are pulled in by an attracting force, they may end up congesting that portion of the map and create a bottleneck for other units passing by.

A more applicable method for tile based games is the use of influence maps [Bough et al., 2004]. These maps are very similar to the graph the pathfinder uses, with the exception that the weight of the edges between every node is initialized to 0. Dynamic events increase or decrease these values to represent the changing aspects of the map. These values are then added to the cost of travelling to that node when its fitness

is examined. There may be multiple layers of such maps to represent different dynamic events. One map may include the changes in weather, like snow or rain causing terrain to be more difficult to traverse. Such changes, while temporary, do not necessarily need to be updated each game step. On the other hand, representing threats from mobile entities will change very frequently based on their current position and orientation, and consequently they would have to be updated (or reinitialized and repopulated) on each game tick. FIGURE 11 displays the threat represented by monsters' field of vision. While this method generates considerable overhead both in storage and processing (may not be efficient for large maps), it works well to simulate a dynamic environment, and will be used in the demo for this project. Populating the influence map can also check if the player is in line of sight of the any of the monsters, and this may initiate a chase reaction.

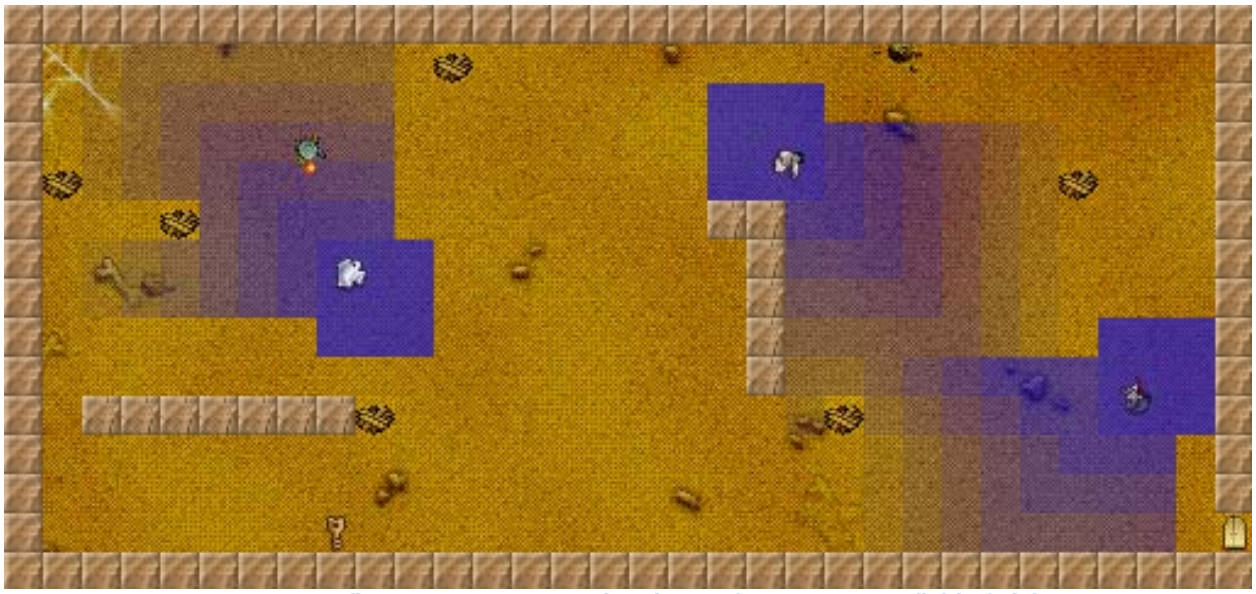

FIGURE 11: Influence map representing danger from monsters' field of vision

4.3 Pathfinding versus navigation

When people use the term pathfinding, they usually refer to an algorithm which is used to move an agent in the world. In a static game environment, pathfinding is indeed the algorithm used to control the NPC. As long as the terrain does not change, the path only needs to be computed once, and the agent can traverse it. The algorithm does not need to keep track of the player's position. This is also true for pathfinding algorithms in dynamic environments. Algorithms like LPA* [Koenig et al., 2004] maintain the shortest path between the two stationary points of interest in a dynamic world. If the player veers off the shortest path for whatever reason, the computed shortest path from its start location and goal may become irrelevant, and a new path needs to be found.

Pathfinding still has an important role in a dynamic environment, but not necessarily in guiding the player through each step. In RTS games for example, it is customary to be able to set a gathering location for units from a production building. Keeping an active shortest path between those two points enables units to move to their gathering point and avoid possible obstacles in the process.

The class of algorithms that help to guide the player are referred to as navigation algorithms. Navigation algorithms keep track of the agent's current position, and adjust their start or goal nodes (depending on the direction of the search) accordingly. The concept behind the navigation and pathfinding algorithms may be very similar, as is the case of LPA* and its navigation version D* Lite [Koenig et al., 2002] [Koenig et al., 2005], but implementation details differ slightly.

4.4 A* Variants for Dynamic Navigation

4.4.1 D* Lite

The D* Lite algorithm (pseudocode listed in APPENDIX B is very similar in concept to the A* algorithm. It uses the same *g* and *h* values to represent the distance from start and heuristic value to goal, respectively. Instead of maintaining an open and a closed list however, a priority queue (U) is used for an open list, and there is no closed list. An additional value (*rhs*) for each node is stored representing the one step look-ahead value of g. In other words, it is the distance from the goal to the node from its best fit neighbour found so far.

In order to understand the algorithm, it is important to realize the relationship between *g* and *rhs*. When a node is initialized, both g and rhs are set to an arbitrarily high value. During any time in the algorithm's execution, if both g and rhs are infinity, it represents an undiscovered or an obstructed node. Upon discovery, the node's rhs value is set to the cost of the path from the start to the node through the node it was discovered from, and the node is inserted into the priority queue. This results in an overconsistent node (g(s)) > rhs(s)). An overconsistent node represents a node in the open list in the A* algorithm. When an overconsistent node is popped off the priority queue, its g value is set to its rhs value (line 12) before its neighbours are discovered (line 13). Hence when g(s) = rhs(s) < infinity, the node is equivalent to a node on the closed list in A*. If during the execution, a shorter path is found to a node on the closed list (g(s) > rhs(s)), the node is called underconsistent (line 14). In this case, the algorithm sets g(s) to infinity (line 15) to signify the node being reinserted into the open queue, and all of its

neighbours are checked to see if this change improves their currently stored path (line 16). The summary of these observations is shown in TABLE 2.

D* Lite condition	A* equivalent
rhs(s) = g(s) = infinity (consistent)	Undiscovered or obstructed node
rhs(s) = g(s) < infinity (consistent)	Node on closed list
rhs(s) > g(s) (overconsistent)	Node on open list
rhs(s) < g(s) (underconsistent)	Node on closed list, with a shorter path discovered

TABLE 2: D* Lite and A* node state comparison

The f value (which mainly used to order the elements in the priority queue of A*) is not stored explicitly for each node. Instead D* Lite implicitly orders the elements based on two keys; namely $\min(g(s),rhs(s)) + h(s_{start},s)$ and $\min(g(s),rhs(s))$. The first value is equivalent to f for the A* algorithm, while the second term stands for the value of g. The function CalcKey(s) is used to calculate this key. In essence, the way the open list is handled for A* is the same as the way it is done for D* Lite.

The UpdateVertex(u) function used to determine the best fit neighbour for the node in question (line 06), and to update the queue according to the new state of the node. The node is removed from the queue (line 07), and then (re)inserted if the node is not consistent (line 08). The removal and reinsertion is done in order to make sure the node's position in the queue represents the changed key values. Optimized versions of the algorithm reposition the node without removal and reinsertion.

The main difference between the A* and the D* Lite algorithms is contained in the Main() function. After the initial path is computed, the agent is moved along the path (line 22). Once changes in edge costs are detected, they are updated, and reinserted

into the queue if necessary (lines 26-27), which is followed by making sure all the changes left the queue in a consistent state (lines 28-29).

4.4.2 D* and Focused D*

FD* [Stentz, 1995] algorithm is based on an earlier version of Dynamic A* algorithm [Stentz, 1994]. Both methods use the same concept to update their map representation based on a dynamic world. As it was the case with D* Lite, the reverse direction is used for the search. Examining the algorithms in APPENDIX C and APPENDIX D, the first important aspect to note is the use of the variable h and the function GVAL. With the reversal of the search direction, the meaning behind actual and estimated cost seems to have gotten reversed as well. GVAL is used to obtain the estimated distance between the current node in question and the AI agent, while the h value represents the known distance from the goal to the current node.

Each node maintains a k value representing the optimal cost found to that node from the goal position since the node was discovered. The k value is used to classify nodes as LOWER or RAISED state. Lower state nodes are the ones with matching k and h values. These nodes are nodes with no obstruction between them and the goal; assuming the nodes were once reachable along the shortest route from the goal since they were first examined. Raised states are nodes from where a detour would be required to reach the goal node due to dynamic obstacles in the way. The k value is also used to order nodes on the open list in a increasing order for the D*, and as a tie breaker for the Focused D*. The rationale behind the idea will be discussed later. Both algorithms only take action when a closed node is modified or the player moved.

Undiscovered nodes or those on the fringes of the search space will not have a direct influence on the calculated path, and hence ignored.

The main difference between the two algorithms is the way they determine what directions to explore. The D* algorithm can be considered a Dijkstra's algorithm in reverse direction that can deal with dynamic terrain changes. The focused D* improves the original version by adding an A* like heuristic to help focus the direction of node expansions.

4.4.2.1 Initialization

Each node when initialized is set to the maximum values. For instance the value used to represent the walls. In that case however, care must be taken not to allow the algorithm to explore wall tiles (or filter out invalid wall paths afterwards), since if the path is blocked, it will find a path through one of the walls. The goal node is initialized to 0 distance travelled and inserted into the heap.

The main method performing most of the calculation is PROCESS-STATE, which processes one state each time it is called. The method is called repeatedly until either the open list is exhausted, or there are no nodes on the open list with lower estimated path to the agent than one that currently exists. The basic idea behind the method is explained next, with {startline-endline} referring to the pseudocode of D* in APPENDIX C, and |startline-endline| to the FD* in APPENDIX D.

4.4.2.2 PROCESS-STATE

The best fit node (which will be referred to as current node) is popped off the heap, or function returns if heap is empty in {1-3}, |1-3|. If node is RAISED state {4-7}, |4-8|, the node's current path to suboptimal and its neighbours are examined to see if a better path is available; if one is found the node is updated accordingly. This action needs to be performed before the next step in case the better path turns out to be an optimal path, thus changing the node to LOWER state. If node is LOWER state {8-13}, |9-14|, neighbour nodes are checked for any unexplored neighbours, non-siblings with potential path upgrades through the optimal node, or if any of its siblings have outdated cost that do not reflect the current conditions. Such nodes are inserted into the open list with updated values for future expansion to propagate the changes. Only optimal nodes are allowed to change non-siblings' values. This is done to avoid creating loops in the search tree. If the current node is still RAISED, its siblings are checked for inconsistencies and inserted into the heap with corrected values {15-18}, |16-19|. If a RAISED state can upgrade a non-sibling's path, the current node is reinserted into the heap to wait for it to become optimal again before making any modifications to the neighbour {20-21}, |21-23|. If a suboptimal non-sibling can upgrade the current node, the neighbour node is inserted into the heap {23-25}, |25-28|, and once again it will have to wait until it becomes optimal before being allowed to change non-siblings. The function returns the new best fit node after the modifications, which will be explored next if it matches the conditions mentioned in previous paragraph.

4.4.2.3 Helper functions

There are various helper functions used by the pathfinder to perform simple operations. Both algorithms use DELETE(X) to delete a node from the heap and sets is state to CLOSED, INSERT(X, newh) to update the key values for the node based on the parameter provided and its current state, and then to insert or reorder the node into the heap. MODIFY-COST() is used to process the changed nodes.

Additionally, the FD* algorithm uses MIN-STATE() returns the minimum node in the heap. Note that the root of the heap is out of date (the agent has moved since the node was last inserted into the heap), it may not be the minimum node. The function GET-STATE() is used by the MIN-STATE() to obtain the node at the top of the heap, which is either returned to MIN-STATE(); or if the node is out of date, its key values are updated, the node is reinserted into the heap, and the next minimum node is fetched. This process continues until the first up to date node is found. MIN-VAL() return a vector of values for the minimum node (obtained by MIN_STATE()) used to compare nodes. LESS and LESSEQ determine if one node is smaller than or smaller than or equal to another node based on the vector of values used for node comparison. GVAL(X) is the estimated distance to goal from node X, and COST(X) is the estimated path length to goal.

4.4.2.4 D and FD* Node Expansion*

As mentioned before, the method of node expansion is what mainly differentiates D* from FD*. D* stores the nodes in increasing order of their k value. OPEN nodes with shorter optimal paths from goal will be checked first even if their current path from goal more costly than other nodes' on the list. This will allow nodes closer to the goal to

establish parents towards the goal first, before other nodes select it as parent; only to have to be reevaluated later when the closer node finds a better parent. The node expansion however is concentric in all directions originating from the goal.

FD* takes a different approach of sorting the open list to better focus the node expansion toward the AI agent. FD* keeps track of an estimate of distance travelled by the agent from the starting position and each node keeps track of where the agent was when the node was last updated (inserted into the open list). It also stores the f value (f=g+h) identical to A*'s method to estimate the distance to the agent's current position. The idea is to expand nodes toward the agent with higher priority than those away from it, in a similar manner to A*'s heuristic expansion. If the player takes two steps before a node is (re)inserted into the open list, its f values will be an estimate to the player's previous position. To keep the values consistent, any time the node's initialized position does not match the agent's current position, the estimated discrepancy between the two positions is added to an outdated node, which is then reinserted into the heap.

As a direct consequence of having an open list with out of date values, when the minimum value is popped off the list, it has to be checked to see if its last updated position matches the player's, and if not, it will have to get updated and reinserted into the list. The simple remove top operation evolved into a popping off multiple nodes off the heap and reinserting them into the heap for resorting. While the FD* examines fewer nodes during expansion in most cases, it performs more operations on them.

5. Implementation Details

In order to be able to compare pathfinding or navigation algorithms, a suitable testbed needs to be developed. My supervising professor, Michel Barbeau, has provided me with a rudimentary game engine, which was capable of loading a world consisting of tiles and a player location, and was able to display it on the screen. Keyboard inputs for moving the player around were also captured and implemented, as well as the collision detection between the player and the tiles. The player moved around the world freely, independent of tile positions, since beyond loading the world and displaying the sprites for the tiles, grid locations were not implemented. The game itself is an Indiana Jones theme two dimensional maze game. The objective of the player is to retrieve a key, and use it to exit the level, while avoiding various monsters roaming the map.

5.1 Map Design Decisions

First thing that needed to be done was to select a method to partition the map into a form that could be represented by a graph. I have chosen to use grid maps, since they require the least amount of fine tuning and pre-processing effort, which were not part of the focus of the report. It is also the method which allowed convenient and fast modification of the map to be able to test various aspects of the project. The initial game engine designated letters A-T to represent different tiles, and ‘o’ indicated the players start location.

To be able to utilize the algorithms, the players needed goals to reach. I have added a set of keys and doors as their objectives. The letters ‘x’, ‘y’, ‘z’ are used for keys, and

‘X’, ‘Y’, Z’ are the doors for their respective keys. An objective comparison between the algorithms requires all the entities to have the same goals. For this reason, once a key is obtained by one player, it remains there for the next player to pick up as well. It can be thought of as the first player with the key locks the door behind him after passing through. At least one matching pair or key/door positions is required to give the players an objective. The placement of the keys and doors is not double checked during the parse, and it is left to the “level designer” to make sure the sequence of key and door positions are obtainable from the players starting positions. The NPC players can be placed on the map by using the letters ‘p’, ‘q’ and ‘r’ to represent one, two or three NPC players starting from the same location. The number of NPC players is limited to three due to screen size restrictions, but it can consist of three ‘p’s, one ‘q’ and one ‘p’ or one ‘r’.

Simulating a dynamic environment can be achieved in a number of ways. Initially I have chosen to add roaming monsters to the maze, their starting positions are indicated by letters from ‘a’ to ‘j’ in the map file. Later on, I have decided against using the monsters in performance tests, as chase and evade behaviors may negatively influence the subjectivity of the experiment. Instead dynamic obstacles are used which represent no threat to the player. In addition to dynamic obstacles, the numbers ‘2’-‘9’ can be used add terrain which is more difficult to traverse, and entities will slow down proportionally when encountering such grids. Blank maze spots in the file are interpreted as grids with a weight of one. TABLE 3 summarizes the convention used for creating maps.

To replace the monsters as the source of dynamic changes in the map, I have selected pit traps appearing on random locations on the map. The traps appear for a random

duration between specified values, making the tile untraversable. Only a certain amount of traps can be present on the map at any given time, as the old traps' timers decay, new ones are added. No traps appear on any nodes which has monsters or players present, mostly to avoid players falling into a pit and getting them stuck for a duration while their pathfinding statistics accumulate.

Any map entry requiring image files are also listed in TABLE 3. All the images need to be located in the /images subfolder (with the exception of the background, which should be placed in the root directory of the project. When the image files are loaded, it is done so in alphanumeric order until the first missing file of the type of images loaded. At that point it is assumed that no other images of that type are present, so with tile_A.png, tile_B.png and tile_D.png present, only the first two will be loaded. Any images representing entities which can face different directions, one image file is required for each direction. The * in the table denotes these suffixes, which are as follows: up, down, left, right, upright, upleft, downright, downleft. The images for the sprites are resized automatically in order to best fit the map onto the screen.

Character	Game Interpretation	Required Image Files
'A'-'T'	Static obstacles such as walls	tile_A – tile_T.png
'a'-'j'	Monster types	monster_a_*.png – monster_j.*.png
'o'	Player controlled character's starting position	explorer_*.png
'p'-'r'	1-3 AI controlled characters' start positions	explorer_*.png
'x'-'z'	Keys	key_x – key_z.png
'X'-'Z'	Doors	Door_x.png-door_z.png
'2'-'9'	Static terrain condition	
' ' or any other character	Basic terrain with a weight of one	

TALBE 3: Character used for map creation

5.2 Non-Player Characters

Since the algorithms work with grid positions, the NPCs need to be able to move between tiles instead of pixels. The simplest way to implement it would have been to increase the timer between the move phases and draw the characters tile by tile. Instead I have chosen to keep the smooth motion that was present in the original project for the player controlled character and build a tile navigation layer on top of the existing pixel one. When an entity reaches the tile in a movement cycle, it requests the next tile to step on, and spends it unused movement distance for that cycle towards the new tile. Each moving entity (NPCs and human controlled player) can move at different speeds.

Before I decided to forego the idea of monsters interacting with players during test runs, I have added some basic functionality for them. Each monster has a 90 degree field of vision, and a modifiable distance to which they can see. The “seen tiles” are stored and was meant to represent the threat to the player from being in the monster’s field of vision. Tiles further away from the monster are assigned lower values than ones closer. Each monster has an aggressiveness value (ranging from 0 to 100), which determines how often they will decide to chase the player when he is visible to them. The path to the last seen player’s position is remembered in case after the next step, the aggressiveness random value doesn’t work out in the monster’s favor and it is forced to choose a random direction. If subsequent aggressiveness rolls are successful, the monster will try to head back to the same place where the player was last seen (assuming a player is not currently seen). When the last seen position is out of line of sight when the monster finishes its random movements, the monster forgets about the path. The monsters in the game do not use pathfinding or have knowledge of the

player's location until the player is spotted. The finite state machine determining their action is shown in FIGURE 12. Images used for monsters were obtained from [Sims2, 2006]

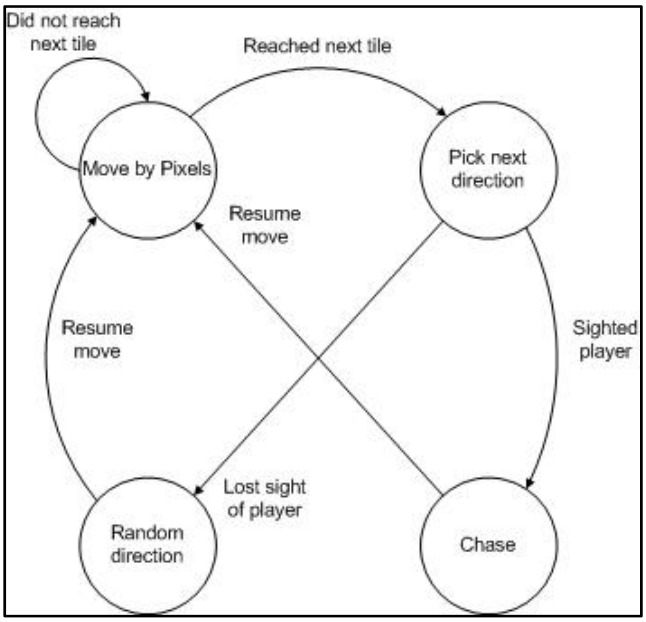

FIGURE 12: Monster behavior Finite State Machine

Upon reaching a new tile, the monster will request a new tile to travel to. The simple equation: $\frac{prob_d * dist_d}{\sum_{d \in D}(prob_d * dist_d)}$ determines the new direction, where $prob_d$ stands for probability of choosing a given direction based on current direction (straight ahead is defined as 37%, diagonally ahead or to either side is 15%, and in the directions behind is1%), and $dist_d$is the amount of tiles visible in the given direction (up to the monster's vision length).

When the player is caught by a monster he is respawned at the starting point. Starting points are designated as safe spots, to which monsters cannot enter to avoid death loops. Key positions are also considered safe. Monsters cannot open doors and

neither can the players who do not have the appropriate keys for them. With the decision to not have monsters affect the experiment results, an option was added (and enabled by default) to make players invulnerable.

5.3 User Interface

At the suggestion of the supervising professor, I have added a separate screen for each NPC player traversing the maze, which is the main reason why the number of NPC players was limited to three. The screens share everything other the information relevant to the specific player and its search engine. The 'g' key is used to display the paths to goal for the NPC players, 't' draws the search trees, and 's' shows the state of the nodes. The threat represented by monster's field of vision, and the monsters' chase paths can be toggled with the 'w' key. Game is paused with the SPACE bar, and only the monsters can be stopped by using 'p'. Players are set invulnerable by pressing the 'i' button. Each of the options listed are toggled, and pressing them again resets previous state. TABLE 4 lists the keyboard shortcuts available.

Shortcut	Action
'g'	Toggles the display of player's path to goal
's'	Toggles the display of nodes' search states
't'	Toggles the display of the search trees generated
'w'	Toggles the display of the threat from monsters
'p'	Toggles pause for monster movement
'SPACE'	Toggles pause for the game
'i'	Toggles players' invulnerability state

TABLE 4: Game key shortcuts

In addition to the three screens showing each players progress through the maze, a stat tracker screen is also present to display a tally of the statistics gathered while achieving

a goal, as shown on FIGURE 13. When the player finishes the level, the figures are added to present a final sum.

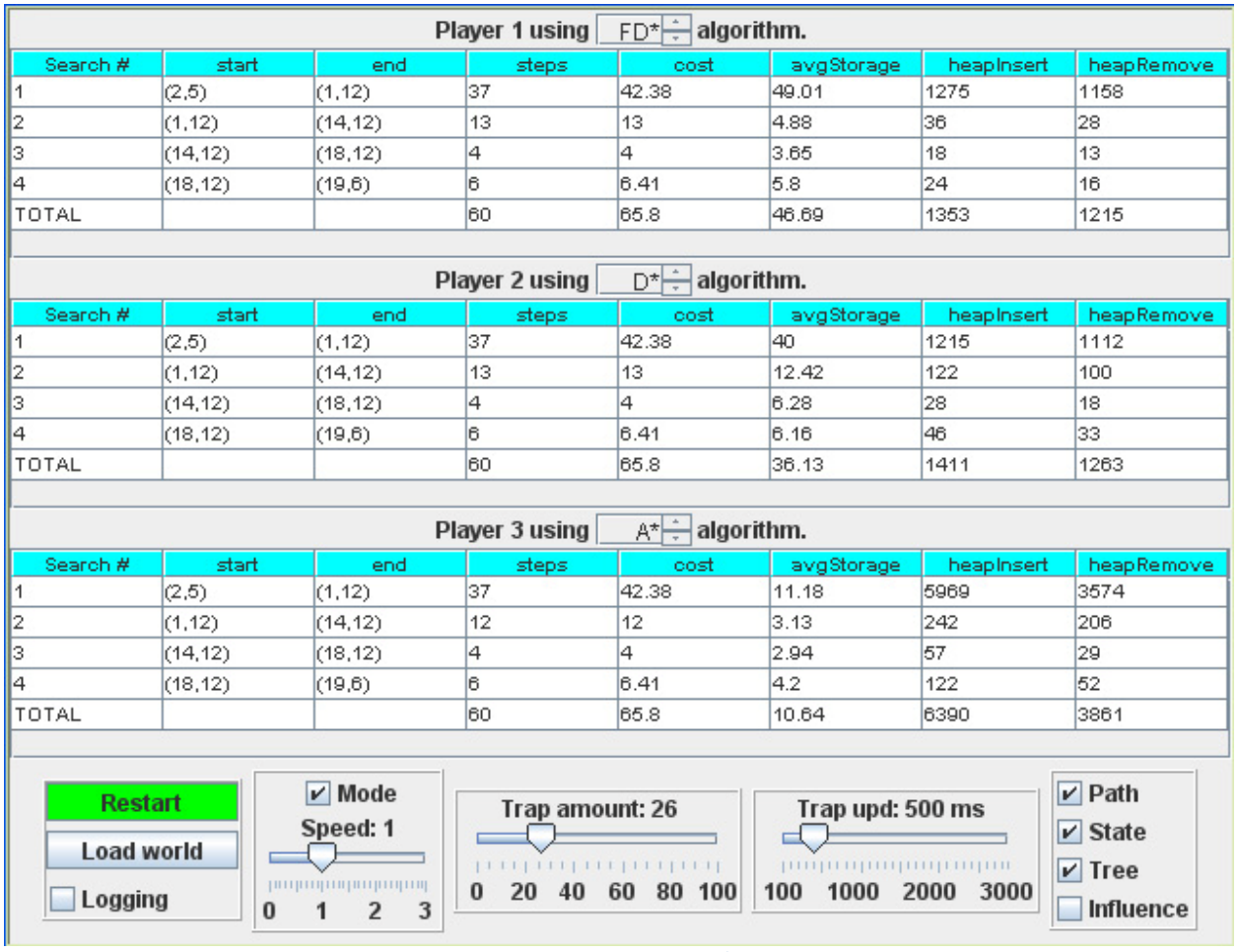

FIGURE 13: Statistics window

The bottom portion of FIGURE 13 contains the control panel which enables modifying the most settings of the experiment. When checked, the Logging option generates the log files for each player upon completion of the level under the /logs directory in tab delimited format. Due to the lack of space on the UI, the term mode was used to refer to the player's path following behaviour. When it is checked, the player will move along an obstructed path (indicated by yellow path) until it reaches the obstacle (at which point the path turns red if no alternate unobstructed paths are available). If the option is not

checked, the player will stop as soon as the path becomes obstructed and waits for an unobstructed path to be found (indicated by green path). The speed slider proportionally modifies the speed of all the entities in the world. The trap amount slider enables the user to select the percentage of the map to be obstructed by dynamic obstacles, while the trap frequency sets the time interval between trap updates. The checkbox group on the right toggles displaying of various search related outputs as described in the previous paragraph. The spinner in the title of each table can be used to select the algorithm to be used by that player. Once the simulation is started, the algorithm selection, and the load world option is disabled until all the players reach their final goal.

5.4 Pathfinding / Navigation Engines

The term pathfinder is used to refer to both pathfinding and navigation algorithms in this section. Each player has its own pathfinder instance computing paths for him. The original intent was to have pathfinders run on their own threads and keeping the world and the player synchronized with them. However, implementation resulted in synchronization issues with three threads waiting on the same events, causing one or more of the threads to miss world updates. Putting the pathfinders back on the main thread resolved the problem, as the game is not allowed to update until pathfinders finish processing the changes.

The pathfinders are structured very similarly. After the ResourceManager finishes parsing the world file, the players are assigned the same sequence of goals. Each

player instantiates their pathfinders, and the initial path calculation begins. When the dynamic objects are added or removed from the world, or the player reaches a new tile, the pathfinders notified by updatePits() and updatePlayers() functions respectively, and necessary adjustments are made depending on the algorithm. Upon completion of the updated path (if one exists), it is sent to the player, who will update its path to goal. Once a player reaches a goal, the node is removed from the goal list, and the same process is repeated for the next goal on the list until the list is exhausted. The significant message sequences are illustrated in FIGURE 14.

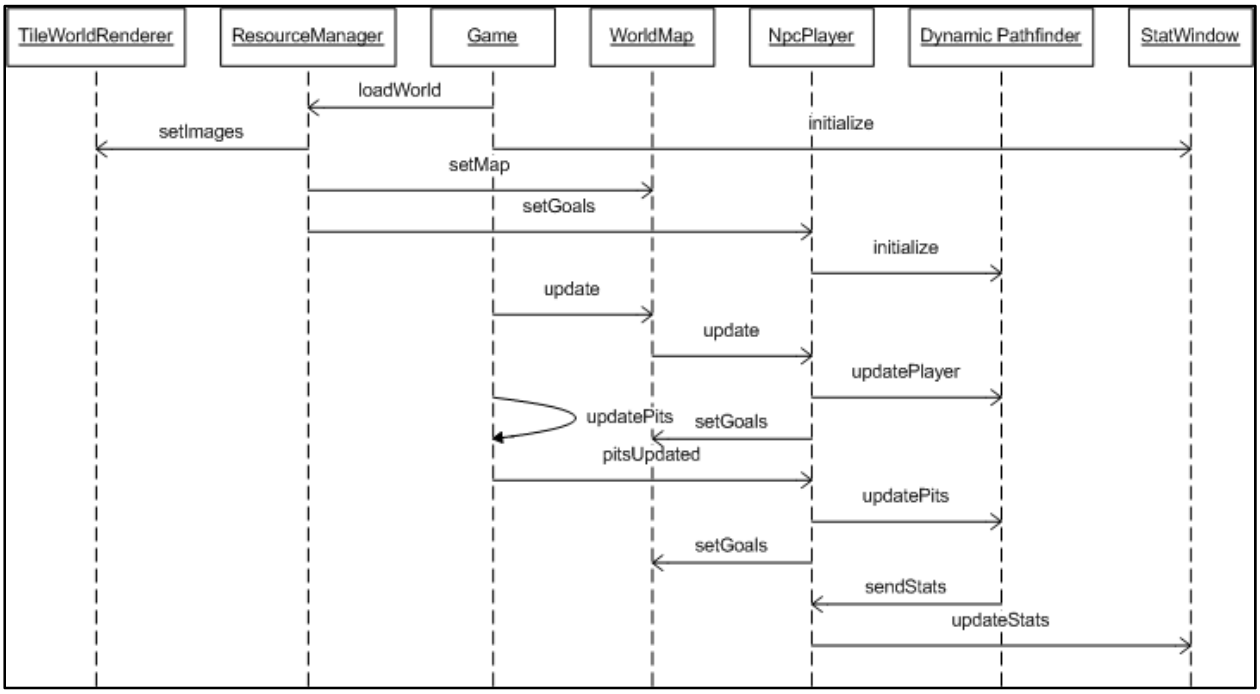

FIGURE 14: Message Sequent Chart involving the significant actors in the project

The A* pathfinder recomputes the entire path on every time the world changes, while the D* algorithms only update relevant portions of their respective maps, namely the closed nodes (if any of them were effected).

All three pathfinders use heap as their open list, and there is no closed list. The A* algorithm maintains the closed list in order to be able to determine if a previously explored node needs to be reinserted into the open list when a more efficient path is found to it. To make the comparisons between the algorithms more consistent, I have modified the algorithm based on the method used for the D* algorithms. Each node keeps track of their states (NEW, OPEN or CLOSED), enabling O(1) existence check for a node in either list.

5.5 Relevant Variables for Testing

Due to the lack of space on the UI, some of the settings cannot be changed at run time. diagonalPenalty in WorldMap class can be set to false if the diagonal movements only cost 1 unit movement instead of its default value of $\sqrt{2}$. Grids can be displayed by setting TileWorldRenderer.drawGrids to true, and background drawing is skipped if TileWorldRenderer.drawBackground is false.

5.6 Adding New Pathfinders

A new pathfinder algorithm can be inserted into the project without much difficulty. First the node used to store results during the search needs to be created. The new node class needs to implement the Comparable interface, since that is the condition the heap uses to order its elements.

The pathfinder class needs to subclass the abstract DynamicPathfinder class and override its abstract methods. The draw methods can be left empty; it will only result in the drawing options for the algorithm to ignore the operation. pathToGoal() is used to create the best path found, which is returned to the player. initialize() resets the

variables used by the search engine, and was mainly added for the event when players were killed to start the search again from their respawn position. The initialSearch() function performs the initial search when the simulation starts or a goal is reached. updatePits() and updatePlayers() signal the pathfinder that actions may need to be taken due to changed environment. When processing obstacles added or removed in the updatePits(), the variable Game.worldMap.getChangedNodes() can be used to access the modified map positions. While it does not matter for static algorithms, since the search is reinitialized at each update event, but dynamic pathfinders should always find a path to the tile the player is moving towards, and not its current position (since the position only gets updated once the player reaches the new tile).

The new algorithm name (preferably an abbreviation of it, so it can fit on the spinner model) should be added to the NPCPlayer.SEARCH_METHODS, and an extra if condition should be created for that name in the createNavigator() method to instantiate the Pathfinder if it is selected.

6. The Experiment

6.1 Experimental Method and Justification

The main purpose of the experiment is to figure out how much faster dynamic algorithms run when compared to their static counterparts. Given that the use of static algorithms is not feasible for large scale unit movement without significant modifications [Higgins, 2002b], and in conjunction with other group movement methods; it would be useful to find out how applicable dynamic algorithms borrowed from the field of robotics are.

With the lack of professional memory monitoring tools, any kind of timing or memory results would not yield any result of consequence. This is especially true with the current scenario, where two or three algorithms are competing head to head within the same application. Instead of gathering data I am currently not available to collect properly, I decided to tally up the number of potentially non-linear function calls for each sub path between the goal locations. In addition to the number of function calls, the average size of the queue for each path segment is recorded as well. Recording these results should give an implementation independent result, which mainly focuses on the algorithm's theoretical running time. The information can also be used to determine the ratio of push and pop operations for instance, which may help in deciding which data structures to use.

In order to test the algorithms in different scenarios, the tests will be conducted multiple times on two different maps. The first map is the maze appearing on the screenshots throughout the document. Moderate amount of traps are used, given the amount of narrow tunnels, where one trap can block off the route to goal on the top and the bottom of the map. Such a scenario would give the A* algorithm a significant disadvantage having to conduct an exhaustive search on the available tiles every time traps are changed. The second map is a large open map, with no walls other than the border of the world. This should give a good comparison between the concentric node expanding D* versus the focused search direction of FD*. The results for each map are tallied and compared to see how each performs in the different scenarios.

6.2 Experimental Results

The experiment was executed on both maps (world2.txt and world3.txt), with the three agents 30 times. The results are included in the /logs subfolder, and are summarized in TABLE 5 for world 2 and TABLE 6 for world 3.

	heapInsert	heapRemove	Average heap size	Total Operations	Average Heap Height	Worst case search depth
Player 1 (FD*)	1289	1200.6	20.97	2489.6	4.39	10929.34
Player 2 (D*)	888.6	809.23	13.86	1697.83	3.79	6439.62
Player 3 (A*)	8522.93	5865.8	7.39	14388.73	2.89	41519.75

TABLE 5: Average search statistics gathered from 30 executions of world2 map

	heapInsert	heapRemove	Average heap size	Total Operations	Average Heap Height	Worst case search depth
Player 1 (FD*)	1480.76	1087.2	63.99	2567.96	6.00	15407.2
Player 2 (D*)	3745.98	3157.17	62.1	6904.15	5.96	41112.7
Player 3 (A*)	17238.53	7599.5	22.72	24838.03	4.51	111917.5

TABLE 6: Average search statistics gathered from 30 executions of world3 map

6.3 Observations

The results for both set of experiments were predictable. The maze environment of world2 map negates most of the advantage of the focused search, and the two dynamic

algorithms expanded essentially the same number of nodes. The overhead of all the node updates due to the player's movements became a significant performance slowdown, which resulted in a 41% decrease in efficiency. As expected, the A* algorithm had to recomputed too much information every step, as was the case with FD*, the entire traversable area had to be examined due to blocked pathways to goal.

The experiment conducted in the open terrain enabled both FD* and A* to take full advantage of their heuristics. Instead of the 700% effort the A* invested into pathfinding when compared to the D* algorithm in the previous test, the difference in the second experiment was significantly reduced. The results prove that while heuristic pathfinders are efficient when used in the right situations, but in worst case (in closed maps, with lot of turns and narrow pathways), they can degrade to a near exhaustive search.

7. Conclusion

7.1 Summary

While the dynamic algorithms showed significant improvement over the static version (especially the FD* on the open terrain), while running the tests my CPU frequently hit 100% usage with just the three agents pathfinding. In games where the computer is responsible for moving hundreds of units around, attempting to use pathfinder for each agent is just not feasible. Expensive pathfinding algorithms should be avoided when it is possible, and other approaches mentioned earlier should be used instead.

7.2 Future Improvements

There are a significant amount of improvements that could have been made, time permitting. Traversable obstacles that only slow travel would have also been beneficial for the experiment. Everything needed to support traversable dynamic obstacles is implemented in the code. A separate influence map is reserved for it, and the game engine checks this map as well during execution, except it is not currently populated. While monsters are not relevant for this experiment for reasons described earlier, since they were mostly implemented, it would have been interesting to add some kind of hide or flee modes to the players. It would also have been interesting to see the pathfinders' performance in a hierarchical architecture or possibly point of visibility method, where the size of the graph is significantly reduced.

I have also spent considerable amount of time trying to get LPA* algorithm working, unfortunately due to the nature of the algorithm, and the denseness of the traps in the trials, the search tree generated included cycles too often to be of any use for testing purposes. My attempts to manually break the cycles created odd looking search trees, which in turn resulted in far from optimal paths. Different tie breaking strategies could have been used as well, to make the trees appear less uniform.

There is also a certain condition under which the D* algorithms enter an updating loop. It happens when a branch of the search tree gets separated from the root and the remaining nodes try to offset the 9999 weight created by the obstruction by continuously inserting each other back on the queue. Such incidents occurred during testing as well, but those results were discarded as irrelevant. The number of operations was in the order of one million on the small map, while the A* algorithm completed the same

search in slightly over ten thousand operations. I cannot be absolutely certain if the problem is related to the algorithm or the implementation, but since the error occurs after map modifications begin, and I had to modify the MODIFY-COST() method, it may be that under some conditions, the updates are done incorrectly. The original version of MODIFY-COST caused update loops as well, but in addition that did not update nodes from which obstacles were removed unless the path to the goal became obstructed, resulting in previous paths going around the obstacle towards the goal to ignore the shortcut created by the newly unobstructed tile.

References:

[Anon, 2003] Anon (2003) Pathfinder Demo, http://www.ai-blog.net/archives/000091.html

[Botea et al, 2004] Botea A., Müller M., and Schaeffer J. 2004. Near Optimal Hierarchical Path-Finding, Journal of Game Development, Volume 1, Issue 1, pp. 7-28.

[Bough et al., 2004] Bourg, D., Seemann, G. (2004) AI for Game Developers, O'Reilly Media, Inc., Senastopol, CA

[Cain, 2002] Cain, T. (2002) Practical Optimizations for A* Path Generation, AI Game Programming Wisdom, Charles River Media, pp. 146-152.

[Coleman, 2007] Coleman, R. (2007) Operationally Aesthetic Pathfinding, Proceedings of the 2007. International Conference on Artificial Intelligence, CSREA Press, pp. 159-163.

[Dechter et al., 1985] Dechter, R., Pearl, J. (1985) Generalized best-first search strategies and the optimality of A*, Journal of the Association for Computing Machinery, pp 505-536.

[Dijkstra, 1959] Dijkstra, E.W. (1959) A note on two problems in connexion with graphs, Numerische Mathematik, Volume 1, pp. 269-271.

[Felner et al., 2004] Felner, A., Stern, R., Ben-Yair, A., Kraus, S., Netanyahu, N., (2004) PHA*: Finding the Shortest Path with A* in an Unknown Physical Environment, Journal of Artificial Intelligence Research, Volume 21, p. 631-670.

[Goodrich, 2002] Goodrich, M.A. (2002) Potential Fields Tutorial", http://students.cs.byu.edu/~cs470ta/readings/Pfields.pdf

[Hart et al., 1968] Hart, P.E., Nilsson, N.J., Raphael, B. (1968) A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Transactions of Systems Science and Cybernetics, Volume 4, Issue 2

[Higgins, 2002a] Higgins, D. (2002) Generic Pathfinding, AI Game Programming Wisdom, Charles River Media, pp. 114-121.

[Higgins, 2002b] Higgins, D. (2002) How to Achieve Lightning-Fast A*, AI Game Programming Wisdom, Charles River Media, pp. 133-145.

[Isla eta al., 2002] Isla, D., Blumberg, B. (2002) New Challenges for Character-Based AI for Games, AAAI Spring Symposium on AI and Interactive Entertainment, Palo Alto, CA.

[Koenig et al., 2004] Koenig, S., Likhachev, M., Furcy D. (2004) Lifelong Planning A*, Artificial Intelligence Journal, 155, (1-2), 93-146.

[Koenig et al., 2002] Koenig, S., Likhachev, M., (2002) D* Lite, Proceedings of the AAAI Conference of Artificial Intelligence, pp. 476-483.

[Koenig et al., 2005] Koenig, S., Likhachev, M. (2005) Fast Replanning for Navigation in Unknown Terrain, Institute of Electrical and Electronics Engineers, Transactions on Robotics, Volume 21, Number 3, pp. 354-363

[Korf, 1985] Korf, R.E. (1985) Depth-first iterative-deepening: An Optimal Admissible Tree Search, Artificial Intelligence archive, Volume 27, Issue 1, pp 97-109.

[Matthews, 2002] Matthews, J. (2002) Basic A* Pathfinding Made Simple, AI Game Programming Wisdom, Charles River Media, pp. 102-113.

[Nareyek, 2004] Nareyek, A. (2004) Computer games - Boon or bane for AI research?, Künstliche Intelligenz, Volume 1, pp 43-44.

[Orkin, 2002] Orkin, J. (2002) 12 Tips from the trenches, AI Game Programming Wisdom, Charles River Media, pp. 29-35.

[Patel, 2000] Patel, A. (2000) Amit's A* pages, http://theory.stanford.edu/~amitp/GameProgramming/

[Reynolds, 1999] Reynolds, C. (1999) Steering Behaviours for Autonomous Characters, Game Developers Conference, San Jose, CA.

[Sims2, 2006] Sims2, (2006) http://www.simmedia.co.uk/info.php?sm=thesims2&page=guides&guide=supernaturalsims

[Scott, 2002] Scott, B. (2002) The Illusion of Intelligence, AI Game Programming Wisdom, Charles River Media, pp. 16-20.

[Stentz, 1994] Stentz, A. (1994) Optimal and Efficient Path Planning for Partially known Environments, Proceedings of the IEEE International Conference on Robotics and Automation, pp. 3310-3317.

[Stentz, 1995] Stentz, A.(1995) The focused D* algorithm for real-time replanning, Proceedings of the 14th International Joint Conference on AI, Montreal

[Tozour, 2002] Tozour, P. (2002) The Evolution of Game AI, AI Game Programming Wisdom, Charles River Media, pp. 3-15.

[Vincent, 2007] Vincent, S., (2007) Navigation Mesh example, http://www.simonvincent.se/portfolio.html

Appendix A - A* pseudocode [Matthews, 2002]

1) Insert the start node in the open list and fill out values of g, h and f
2) Remove the node N with the lowest weight from the open list, and insert it in the closed list
3) If N is the goal node, return constructed path
4) For each unobstructed neighbor (child) C of N, calculate the g, h and f values
    a) If C is already on the open or closed list, check if the new f is lower than the old value
        i) If it is, update the node's g, h, f and parent values, and propagate the change to reflect a shorter path found through the node C
    b) Else insert node C in the open list and populate the g, h, f and parent values
5) Repeat process from step 2 if the open list is not empty

Return, no path can be found

Appendix B - Pseudocode for D* Lite [Koenig et al., 2005]

```
procedure CalcKey(s)
{01} return [min(g(s), rhs(s)) + h(s_start, s); min(g(s), rhs(s))];

procedure Initialize()
{02} U = Ø;
{03} for all s ∈ S rhs(s) = g(s) =∞;
{04} rhs(s_goal) = 0;
{05} U.Insert(s_goal, CalcKey(s_goal));

procedure UpdateVertex(u)
{06} if (u ≠ s_goal) rhs(u) = min_{s' ∈ Succ(u)} (c(u, s') + g(s'));
{07} if (u ∈ U) U.Remove(u);
{08} if (g(u) ≠ rhs(u)) U.Insert(u, CalcKey(u));

procedure ComputeShortestPath()
{09} while (U.TopKey()<CalcKey(s_start) OR rhs(s_start) ≠ g(s_start))
{10}  u = U.Pop();
{11}  if (g(u) > rhs(u))
{12}      g(u) = rhs(u);
{13}      for all s ∈ Pred(u) UpdateVertex(s);
{14}  else
{15}      g(u) = ∞;
{16}      for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);

procedure Main()
{17} Initialize();
{18} ComputeShortestPath();
{19} while (s_start ≠ s_goal)
{20}  /* if (g(s_start) = ∞) then there is no known path */
{21}  s_start = arg min_{s'∈Succ(sstart)}(c(s_start ,s') + g(s'));
{22}  Move to s_start;
{23}  Scan graph for changed edge costs;
{24}  if any edge costs changed
{25}      for all directed edges (u, v) with changed edge costs
{26}          Update the edge cost c(u, v);
{27}          UpdateVertex(u);
{28}      for all s ∈ U
{29}          U.Update(s, CalcKey(s));
{30}      ComputeShortestPath();
```

Appendix C - Pseudocode for D* Algorithm's PROCESS-STATE() [Stentz, 1994]

```
Function PROCESS-STATE()
L1    X = MIN-STATE()
L2    if X = NULL then return -1
L3    k_old = GET-KMIN();  DELETE(X)
L4    if k_old < h(X) then
L5       for each Y neighbor of X:
L6          if h(Y) ≤ k_old and h(X)>h(Y) + c(Y,X) then
L7             b(X) = Y; h(X) = h(Y) + c(Y,X)
L8    if k_old=h(X) then
L9       for each Y neighbor of X:
L10         if t(Y)  = NEW or
L11                (b(Y) = X and h(Y) ≠ h(X) + c(X,Y)) or
L12                (b(Y) ≠ X and h(Y) > h(X) + c(X,Y)) then
L13            b(Y) = X; INSERT(Y,h(X) = h(Y) + c(Y,X)
L14   else
L15      for each Y neighbor of X:
L16         if t(Y) = NEW or
L17                (b(Y) = X and h(Y) ≠ h(X) + c(X,Y)) then
L18            b(Y) = X; INSERT(Y,h(X) + c(X,Y))
L19         else
L20            if b(Y) ≠ X and h(Y) > h(X) + c(X,Y) then
L21               INSERT(X,h(X))
L22            else
L23               if b(Y) ≠ X and h(X) > h(Y) + c(Y,X) and
L24                     t(Y) = CLOSED and h(Y) > k_old then
L25                  INSERT(Y,h(Y))
L26 return GET-KMIN()
```

Appendix D - Pseudocode for Focused D* Algorithm's PROCESS-STATE()[Stentz, 1995]

```
Function: PROCESS-STATE()
L1    X = MIN–STATE( )
L2    if X = NULL then return NO – VAL
L3    val ≤ f(X), k(X)>;k_val = k(X); DELETE(X)
L4    if k_val <h(X) then
L5    for each neighbor Y of X
L6        if t(Y) ≠ NEW and LESSEQ(COST(Y), val) and
L7              h(X)>h(Y)+c(Y, X) then
L8            b(X) = Y; h(X)=h(Y)+c(Y, X)
L9    if k_val = h(X) then
L10       for each neighbor Y of X
L11           if t(Y) = NEW or
L12                 (b(Y) = X and h(Y)≠h(X)+c(X, Y)) or
L13                 (b(Y)≠X and h(Y)>h(X)+c(X, Y)) then
L14               b(Y) = X; INSERT(Y,h(X)+c(X, Y))
L15   else
L16       for each neighbor Y of X
L17           if t(Y) = NEW or
L18                 (b(Y) = X and h(Y)≠h(X)+c(X, Y)) then
L19               b(Y) = X; INSERT(Y,h(X)+c(X, Y))
L20           else
L21               if b(Y)≠X and h(Y)>h(X)+c(X, Y) and
L22                     t(X) = CLOSED then
L23                 INSERT(X, h(X))
L24               else
L25                 if b(Y)≠X and h(X)>h(Y)+c(Y, X) and
L26                       t(Y) = CLOSED and
L27                       LESS(val, COST(Y)) then
L28                     INSERT(Y, h(Y))
L29   return MIN–VAL()
```

Appendix E - Files included on the disk

ControlPanel.java
Game.java
GameKey.java
KeyListener.java
Heap.java
HeapExceptions.java
Door.java
Entity.java
MapNode.java
Monster.java
NPC.java
NPCPlayer.java
Player.java
WorldMap.java
AStar.java
AStarNode.java
DStar.java
DStarNode.java
DynamicPathfinder.java
FDStar.java
FDStarNode.java
Node.java
SearchStat.java
State.java
Vector.java
ResourceManager.java
Sprite.java
StatPanel.java
TileWorldRenderer.java

image files in /images folder
map files in /worlds directory
log files of the two experiments in the /logs folder

The project's workspace can be imported into Eclipse for examination and execution.