Optimizing modern pathfinding methods

In Imperfect 2D Environments

Luc Shelton
University of Derby
Department of Business, Computing and Law
Derby, United Kingdom
lucshelton@gmail.com

Abstract—**As a common means of generating pathfinding within games today, methods such as A* and greedy best-first search utilize heuristics with path costs to determine early on in the path generation what is the most optimal route. While the end result is most certainly plausible, it can be proven to be somewhat expensive on the CPU to generate within large environment, especially as the game state may change during the traversal between point A and point. As a means of looking further into this issue, I aim to determine whether or not there are methods that can be taken to adjust the already generated paths so that a smooth and less-expensive traversal from two given points can be accomplished.**

Keywords—pathfinding; dynamic; path; D* Lite; A*; heuristics; artificial intelligence; LPA*; games; JPS; Jump-Point-Search

I. INTRODUCTION

Pathfinding within games has long been considered as an area of concern for being a computational bottleneck. This is reaffirmed when AI is required to constantly generate several new path plans in a small timeframe [6]. As a cause for research, we decided to pursue other ideas that would optimize the way that agents moved around a given environment that happens to change every so often. As a test-bed for our research we will be carrying out a series of tests on a game we have developed called of *Ant Frenzy.* Previously we have attempted to utilize path smoothing methods such as Bezier curves [12] in an attempt to cut the path traversal times, and display a natural traversal of the path by the ants in our game. We discovered that in some instances this wasn't necessary considering the effect that the physics had on the ants within the game. Therefore we aim to carry out research into alternate methods that are applied to achieve the generation of a path quickly and accurately. The rules of *Ant Frenzy* is simple, in order to win the game, you must ensure that the enemy ants are not able to eat the cake that is positioned somewhere within the level for a fix period of time. Within this paper, we aim to determine whether it is possible to find a suitable trade-off between computation, memory and traversal times and given the environment in which they are used, which one would be the most appropriate and why. Moreover, we aim to determine whether dynamic re-planning algorithms are more optimal in every situation in which the agent will be forced to update the path.

II. RELATED WORKS

A. A* Pathfinding

Considered to be the most commonly utilized informed pathfinding method in games, this algorithm takes into account three values that determine the strength of nodes during the search process of the path. These are considered to be the G cost (accumulative cost), H cost (heuristic) and F cost (both combined). Furthermore, two lists are used of which one is considered as the closed, and the other being the open. Upon each iteration (until the goal state is found), the eight-neighboring nodes of that one that is being actively observed during the search are placed into the open list and the current node that is being observed is then added to the closed list. The open list is considered to be a list of potential nodes that are to be observed when finding the next node to move onto during the search. The closed list on the other hand is the collection of nodes that we know are going to be used during the path back-propagation process. During each search, the node that has the lowest F cost within the open list is then selected to be the next node that is observed for its neighboring nodes. This loop continues repeatedly until the search discovers the goal node to be one of the neighboring of the one that is being actively searched. The path is then reconstructed through the parent of each node that is stored in the closed list.

It's noted that making use of a priority queue ensures that the during the search procedure, the node with the lowest F score is always placed at the front of the list. This is so that when the *Pop* function is called, it is removed and provided to the user [9]. This means that there is less computation time spent searching linearly through the open list for the node with the lowest F score. Additionally it is also recognized that A* does not make use of any kind of re-planning information and when required to regenerate, it will restart from the beginning.

B. LPA* Pathfinding

Based on the A* pathfinding method, LPA* applies the finite graph problem on known environments where the edge costs (i.e. grid spaces) increase and decrease over time [2]. LPA* will always determine the shortest path from a given

start vertex to a given goal vertex based on the information that is generated from previous planning iterations (previously stored edge costs). The variables that are used during the process are represented by an RHS and G cost. Should the RHS and G cost never equal the same then they are considered as locally inconsistent.

The G cost, just like with A*, is considered as the accumulative cost of traversing from the start to where the search node is actively looking at on the map. Similarly, LPA* also makes use of the Euclidean distance between the current node to the end goal to determine its heuristics value as a part of its RHS value. [2]

C. D* Lite

Sven Koenig [2] proposes a method of pathfinding that combines the functionality of both D* and LPA* dynamic path-planning methods. The main difference between the two algorithms however is that D* Lite takes into account the path that has already been travelled by applying a *K* value when calculating the key of certain nodes. D* Lite was originally introduced for usage in robotics to find the most optimal routes within environments that were considered as only partially visible or otherwise known as imperfect. During the robots traversal through an unknown environment, the agent would concern itself with a surrounding eight-connected graph. Should the robot discover that one of the edges is not traversable, it changes the cost of that particular node to infinity. It was for this reason that information regarding the environment had to be stored to allow for smoother re-planning in the future should the agent (either a robot or in-game AI) discovers an obstacle. The premise of D* Lite works in a similar fashion to A* however there are some notable differences. Should there be no changes to the environment, then it will generate a path just like A* in that it will simply search based on the smallest returned Euclidean heuristics. On the other hand, should there be information stored in regards to the surrounding terrain that the agent is traversing through then, the *ComputeShortestPath* function as stated by Sven Koenig [2] will check for inconsistencies. Every time an obstacle is detected within the terrain, the Open List is updated with locally inconsistent nodes and the *CalculateKey* [2] function is used to determine a tuple value that the priority queue is then ordered lexicographically by. Refer to Fig. 1 to understand the layout of the key.

First

Min(G(x),RHS(x)+H(x)) + k value;

Second

Min(G(x),RHS(s));

Fig. 1. The *CalculateKey* function that is used in the *ComputeShortestPath* for ordering the locally inconsistent nodes in the open list [2]

The premise of the algorithm is that it makes use of a new value called the *RHS* (Right-Hand-Side) coefficient. This value is then used to determine when there are inconsistencies within the finite graph. Determining whether or not there are inconsistencies in the stored finite graph is used when having to re-compute the shortest path by attempting to make use of the path that has already been traversed.

As such, a *K* value is made use of to store the heuristics from the current start node to the goal node. If we are to consider an agent traversing a path, the node that it is currently on is always considered as the start node. Therefore, when we update the start of the path, then the heuristic distance between the new starting point and the current goal is added on to the *K* value. This value is then added to all newly calculated keys for when nodes are added to the open list. This way we avoid having to go through the open list queue every time connections between nodes in our finite graph change for whatever reason.

When the nodes are processed through the open list, their adjacent vertices are updated accordingly based on whether node is considered as *over consistent* and *under consistent*. The key difference between the two is that when a node is considered to be over consistent, it means that there has been discovered to be a shorter path and it requires now updating. Under consistent suggests that a path has been discovered to be blocked and adjacent vertices also require updating too [2].

We aim to determine whether using an algorithm such as this within a game that has perfect information and dynamically moving objects, would be a viable option as well. Additionally we aim to see at what point during the game state that it would be considered as unnecessary.

D. Hierarchical Pathplanning A* (HPA*)

In contrast to the A* pathfinding method, HPA* applies a level of abstraction to the environment that the agent is traversing when determining the best path to take. Taking into account the idea that path information can quickly be invalidated based on what can occur within the game state, the A* method is quickly recognized the waste computation when it has to typically re-plan again shortly after [5]. Instead, HPA* discretizes the environment as linked local clusters so that when path planning has to occurs it is capable of approximating the quickest path most of the time without any real problems. This is achieved through conducting pre-processing before the requesting entity within the game has to plan a path for the first time. It's also stated that should the terrain change at all within the environment, then any of the locally affected clusters of neighbors are updated and the rest of the finite graph remains intact.

It is only once the shortest path is approximated through the usage of the linked neighbours that we then refine the path by using A*. The right side to the image demonstrated in Fig. 1 demonstrates the approximated path that is generated through using the pre-processed linked neighbours. This has been noted as being optimal has agents benefit from not having to waste computation time should the path have to change midway through traversal. Furthermore, agents will immediately have a direction to able to head in within the environment without any further refinement having been done to the path.

Path refinement in this context would consist of translating a path from an abstract level (such as the graph representation of our terrain), and using A* to determine any kind of intermediate collisions in the process of the path generation.

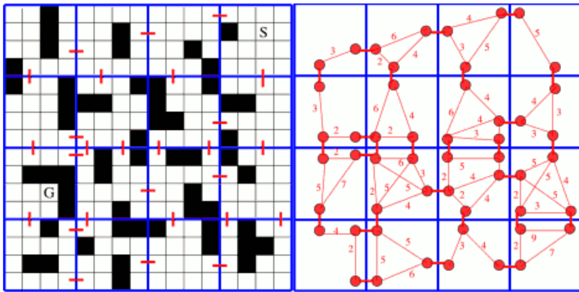

Fig. 2. The environment representation used by HPA* when clustering neighbouring nodes. The S and G lettering denote the start and goal. [5]

E. *Theta* Pathfinding*

Making usage of the same heuristics as A* pathfinding, Theta* utilizes the likes of ray-traces to determine the fastest possible route. A* typically finds the shortest route based on the visible 8 neighboring nodes that we are observing and assigns the parents accordingly. Therefore the end result doesn't necessarily tend to be the shortest version [3]. This problem is then solved by not restricting the node within the search space to having the current node as the parent but instead, is able to have the same parent as the current node. This is so long as it is within the line of sight (through ray-tracing) of the one that is being searched. This combined with a priority queue [8], means that it is capable of finding the shortest path most of the time. As mentioned previously, A* pathfinding constructs its path after it completes its search through recursively going through the parent of each node within the closed list. Considering that these parents can be adjusted based on what is visible through the ray tracing algorithm, this means that the path can be shortened substantially.

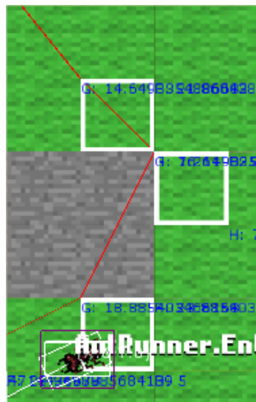

Fig. 3. Line intersections over blocked nodes (gray tiles) determine where to use next neighbour as parent.

Typically the Bresenham line drawing algorithm (Fig. 4) is used to plot points on a grid and similarly this could be seen used in an environment discretized as a two-dimensional array. Refer to Fig. 3 for further illustration of its implementation. This line drawing algorithm is used to determine whether there is an intersection between the neighboring node that we are observing and the currently selected nodes parent that has the lowest F score from the list.

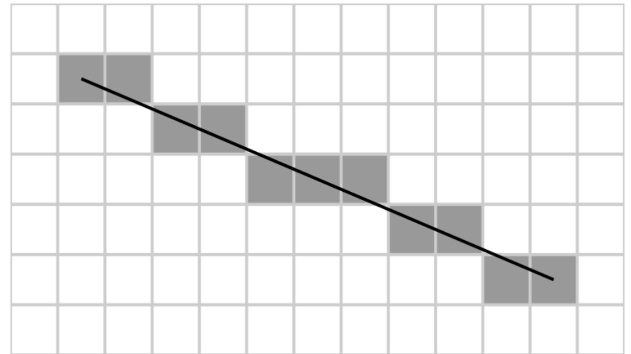

Fig. 4. An image of the Bresenham line drawing algorithm. The cells coloured gray represent the grid within our game that it would intersect with.

F. *Jump-Point Search A**

Based on symmetric pruning previously demonstrated through the methods of RSR [4], Jump-Point-Search (JPS) consistently speeds up the speed in which the A* pathfinding method is performed. This is achieved by neglecting neighboring nodes during the search procedure based on the condition of two simple pruning rules. These rules are applied in both a straight step and diagonal step from the parent of the current node that is being observed. The aim is to recursively prune any neighboring nodes to the current node that are proven to not provide any further optimal path [4]. This is done by determining the direction of travel from the parent node to the one that is being currently observed. The nodes that are then determined as a successor, are placed into the open list and then further evaluated in a similar fashion to that of the original A* algorithm in that the node within the open list that has the lowest F score, is investigated first.

Immediately we can see that this is a huge improvement over the previous iteration of the A* algorithm, in that we don't have to explicitly examine every neighboring node to the current one that we are examining at the time. Instead promising neighbours within the open list are examined recursively jumping in the direction by normalizing the distance between the node we are examining and its neighbor (or successor). This means that for the most part we get rid of wasteful computation by focusing on the ones that are more relevant to the generation of the path itself. This JPS method was bench marked by using map information from released games such as *Baldur's Gate* and *Dragon Age Origins* and as such was recognized to speed up the A* pathfinding algorithm up to 3-26 times depending on the map information that was used at the time. [4]

When compared against Theta*, our alternate method for path shortening, we can argue that there is a significantly small memory overhead in comparison considering that the search procedure itself is preventing nodes from being added to an open list unless it is considered necessary [3][4]. The algorithm itself then becomes competitive with the likes of HPA* in that more times often than not, it will generate a better path. [4]

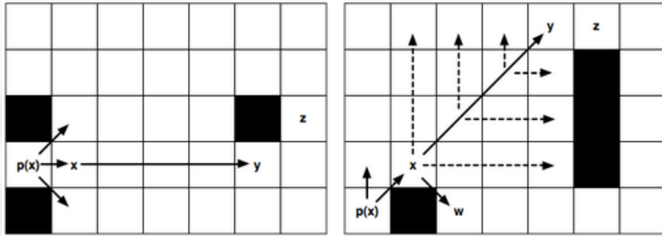

Fig. 5. Demonstration of the path jumping rules in both straight step (left) and diagonal step (right). [13]

Fig. 6. Demonstration of the JPS pathfinding method in Ant Frenzy.

III. Conducted Research

To further understand the cost of utilizing a pathfinding method that handles re-planning, we prepared several separate game environments that that tested our methods of pathfinding through playability, and optimality. To demonstrate, the four methods of pathfinding that we will be applying will be an optimized version of A* pathfinding with priority queues, D* Lite and Theta* to determine which is considered most appropriate for re-planning the shortest paths. The set up for the research will consist of ants navigating within a level prepared using the .TMX file format and the Tiled map editor [4]. The end goal for the ants within our test-bed will be an object resembling a cake.

The first environment consists of a 100x100 grid based game that will deliberately be intended to be expensive on the CPU for the pathfinding algorithms we are benchmarking. From this test, we aim to determine how fast and effective the heuristics from each of algorithms are. Furthermore, there will be objects moving within the environment that will block the path of the ants causing for them to re-plan their paths at a moment's notice.

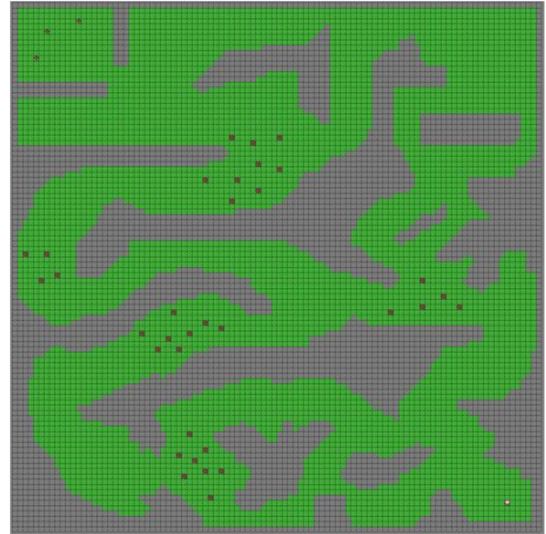

Fig. 7. The level overview of the 100x100 map – The cake is in the bottom right corner of the map while the starting points are in the top left. Notably there are several choke points in the map which should conflict with the ants during traversal.

The second environment resembles the shape of a zigzag – our aim within this environment is to determine whether given the shape, the ants are capable of generating the most suitable and shortest paths to the end goal.

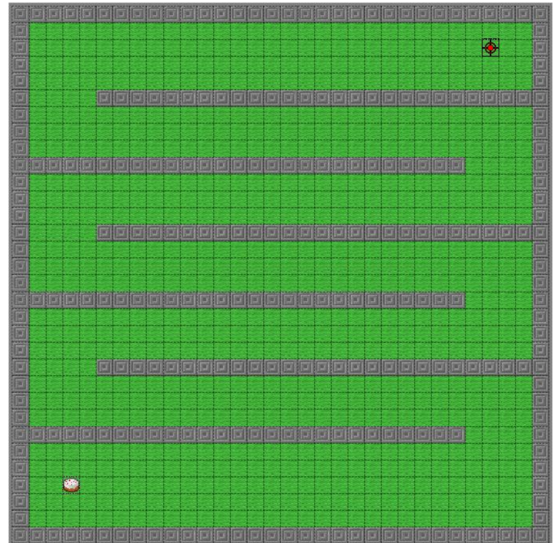

Fig. 8. The level overview of the Zig-Zag map

Each of our environments will contain physical objects that will move at a random interval, speed, and angle providing for an interesting challenge for the pathfinding method of choice by the ants within our game. Should the ants encounter the obstacles during their traversal of the games environment, they will be forced to re-plan immediately with their own respective method of pathfinding. During the research process, we will be actively monitoring the behavior of the ants to deduct if there are any anomalies in the way that they generate the paths and

how the behave in response to dynamically moving objects in the maze.

Lastly, we will spawn 10 separate ants that will make use of their own path-finding information for each of the path-finding methods. Red ant will make use of Theta* pathfinding, Blue ant will make use of D* Lite pathfinding, Yellow ant will make use of JPS enhanced A* path-finding and the Green ant will make use of an optimized version of A* with heap-sorted priority queues.

A. Tools

To conclude the results that we gained from the testing, we made use of a basic click-wheel interface that enabled us to spawn obstructions and ants on the fly within the testing environment.

Fig. 9. Clickwheel interface used for spawning the ants in the environment.

B. Optimisations

1) Theta*

The first problem that we encountered with the Theta* enhanced A* algorithm was that there were certain scenarios in which the pathfinding process would not immediately find the most appropriate path. The problem means that when paths were being generated, the ray-tracing algorithm (line of sight) was being used to determine if there was an intersection between the nodes that we were observing from the open list.

Furthermore we noticed that with the usage of Theta* within a dynamic environment we were unable to determine if the path was blocked by simply checking the future path node that we were heading towards. Instead, to counter this problem we made use of a 64x64 px bounding box (same size as the tiles in the level) that's aligned to the rotation of the ant so that it could detect for incoming collisions based on the direction that it was heading in.

C. Heuristics Generation

1) Manhattan Distance

Alternatively known as the rectilinear distance, the Manhattan Distance is derived from the idea of taxi-cabs traversing across the Manhattan Island. Because of the lattice like nature of the buildings, there are numerous ways in which one could consider the most optimal route from one point to another [14].

Refer to the below source code to see the Manhattan distance.

```
Math.Abs(pA.position.X - pB.position.X) * m_StraightCost +
Math.Abs(pA.position.Y - pB.position.Y) * m_StraightCost;
```

Fig. 10. The code used for the Manhattan distance with a proprietary "straight cost" in C#.

2) Euclidean Distance

Stemming from Pythagoras theorem, the Euclidean distance is also considered as the "true distance", considering that it takes the straight distance between one coordinate to another [9].

Refer to Fig. 11 for the source that was used to acquire the distance using Euclidean metrics.

```
    var _distanceX = pA.position.X - pB.position.X;
    var _distanceY = pA.position.Y - pB.position.Y;

        return Math.Sqrt(_distanceX * _distanceX +
_distanceY * _distanceY) * m_StraightCost;
```

Fig. 11. The code used for the euclidean distance in C#.

Our intent for making use of several means of heuristics is to do with what effect that it has with what kind of difference there is

D. Equations used

1) Bresenham Line Drawing

The Theta* pathfinding algorithm used the means of ray casting to determine whether the eight-way neighboring node of the current node within the search space was capable of reaching the parent of our current node. Referring to Fig. 3, we can see that the nodes that the line covers are colored in a darker gray depicting the nodes within the graph that are detected in the intersection. If we detect an intersection between the node that we are observing and the parent of the current node, then set the parent of the observing node to be the current node. Otherwise, if this is not the case, then the parent for the nod being observed becomes the parent of the current node. Still, we can make use of the same algorithm to determine if there is a collision mid-way through the traversal of the path. Typically if we were to determine that the path generated was not traversable, we could do this through using the line-drawing algorithm and determine at what distance the box is at and then from there generate a new path if it is considered necessary.

E. Pathfinding methods

In our experiments we will make use of 4 different pathfinding methods to determine which provides the most optimality in regards to path generation and traversal times as well as memory overhead.

1) D Lite*

As this method is typically used for robots discovering imperfect environments therefore we will apply the same method with our ants. As soon as the ants discover a part of the terrain that wasn't there during their initial path planning procedure, it will update the pathfinding object, and then compute a new path to head in that avoids the newly discovered obstacle. At the beginning of the game state however, we will inject the information about tiles that are considered not passable within the level. Due to the dynamic moving nature of the boxes however, we shall constantly check to see if the next path node that the agent is heading towards (ants) is blocked by a box. If it is, we will update the algorithm and re-plan accordingly. We will furthermore conduct another test where there will be no further information presented to the agent to make use of and therefore will depend upon enabling the agent to discover obstacles in its path and update the open hash accordingly.

We additionally aim to demonstrate the usage of the algorithm without the pre-requisite of having to provide the terrain information to the agent beforehand. In this sense, it will make use of the eight-connected graph robot navigation strategy that is stated in Koenig's paper [2]. As such, the agent will be forced to learn about the environment as it re-plans around the terrain.

2) A Pathfinding with optimizations*

The optimizations that we aim to carry out with this method is to utilize the likes of a heap sorted priority queue along with other O(1) based operations to speed up the pathfinding procedure [8]. During the search procedure of the algorithm, nodes that are of interest to the agent will be placed into the open list based on whichever has the lowest F value. This means that when it comes to determining the next appropriate node to explore, we can immediately select the one that is at the top of the queue. This overall speeds up the search procedure, as we no longer have to carry out a separate linear search procedure to determine the next node in the list that we will observe.

Heuristic values for each of the nodes within the game state environment are stored within a 2D array so that it is quick to acquire the values that we need when searching within nested for loops (i.e. observing neighbours). We recognize that making use of A* means that it won't get the same re-planning optimizations as D* Lite however, we aim to determine in which circumstances it would be necessary to have an algorithm such as that with a large memory overhead.

*3) Theta** - Our implementation of this algorithm will be combined with the usage of the aforementioned Bresenham line drawing algorithm. We will experiment between the likes of the Manhattan and Euclidean distance heuristics to determine whether or not there is any particular performance impact with the path generation whatsoever.

*4) Jump-Point-Search enhanced A** - As another means of testing aesthetic based algorithms, we decided to put this against the likes of Theta* and even dynamic re-planning based algorithms to determine whether it's really worthwhile basing our computation speeds on the memory overhead of dynamic re-planning methods, or to simply enhance the search process itself. As stated previously within section II, D* Lite is capable of generating a path quicly based on previous iterations of path searches because it reuses the previously generated path.

F. Results

TABLE I. PATH GENERATION TIMES (ZIG-ZAG)

Distance	**Re-planning Times**		
	Pathfinding Method	***MIN***	***MAX***
copy	D* Lite (Full Knowledge)	3260 ms	3260 ms
	A* Optimized (Euclidean)	63 ms	63 ms
	A* Optimized (Manhattan)	32 ms	32 ms
	Theta* (Euclidean)	74 ms	74 ms
	JPS A*	0 ms	15 ms
	D* Lite (Imperfect Knowledge)	0 ms	15 ms

TABLE II. PATH TRAVERSAL TIMES (ZIG-ZAG)

Distance	**Path Completion Times**		
	Pathfinding Method	***MIN***	***MAX***
copy	D* Lite (Full Knowledge)	1 m 41 s	1 m 41 s
	A* Optimized (Euclidean)	1 m 35 s	1 m 35 s
	A* Optimized (Manhattan)	1 m 35s	1 m 35 s
	Theta*	1 m 24s	1 m 24 s
	JPS A*	1 m 33 s	1 m 33 s
	D* Lite (Imperfect Knowledge)	1 m 47s	1 m 47s

TABLE III. PATH GENERATION TIMES (100x100 MAP)

Distance	**Re-planning Times**		
	Pathfinding Method	***MIN***	***MAX***
copy	D* Lite (Full Knowledge)	1795 ms	1815 ms
	A* Optimized (Euclidean)	780 ms	858 ms
	A* Optimized (Manhattan)	905 ms	918 ms
	Theta*	1023 ms	1125 ms
	JPS A*	16 ms	32 ms
	D* Lite (Imperfect Knowledge)	0 ms	15 ms

TABLE IV. PATH TRAVERSAL TIMES (100x100 MAP)

Distance	**Path Completion Times**		
	Pathfinding Method	***MIN***	***MAX***

Distance	Path Completion Times		
	Pathfinding Method	***MIN***	***MAX***
copy	D* Lite (Imperfect Knowledge)	2 m 3 s	2 m 3 s
	A* Optimized (Euclidean)	1 m 45 s	1 m 45 s
	A* Optimized (Manhattan)	1 m 44 s	1 m 44 s
	Theta*	1 m 28 s	1 m 28 s
	JPS A*	1 m 38 s	1 m 38 s

IV. ANALYSIS

A. *Memory Overhead*

As we've stated in the related works, D* Lite makes use of an open hash that stores values of the environment that it navigates through so the re-planning of a path with the algorithm, it will be faster to do than the last time. More notably however, due to the way that it works, whenever a cell has been changed within the finite graph, it then has to be updated through the means of the *ComputeShortestPath* function that is stated by Koenig [2]. We noticed that in some circumstances that here was a marginal FPS slow down when the agent that was utilizing the D* Lite algorithm was traversing the environment. We believe that this has something to do with the way that it is cycling through the open list of inconsistent nodes as it plots the cells that are considered to be impassable. As such, this is something that we will have to look into in our later work.

B. *100x100 Map*

Immediately we can see that the path generation times across all the algorithms that were used are ranging between half a second to just over one second. More surprisingly, we were expecting that the D* Lite algorithm would be more optimal in finding a path within a shorter period of time to begin with while fully informed. We did noticed within testing however that when we did not inform the pathfinding algorithm about the terrain, that it was capable of generating a path much faster. We believe that this is simply down to less nodes have to be expanded during the computation of the shortest path.

When the agent was forced to learn about the environment as stated within Sven Koenig's paper [2] we were seeing generation times ranging anywhere between 0 ms to 15 ms consistently through-out the entire generation process. Admittedly the path traversal was much slower seeing as it had to learn about the terrain first, but it managed to arrive at the goal no longer than 45 seconds more than the other pathfinding methods. In contrast to Theta*, the path traversal time as a whole was slower for D* Lite, but we noticed that it was not of a highly noticeable margin.

More surprisingly however, we were intrigued to see that JPS A* was capable of generating path plans for the entire 100 x 100 map within a competitive margin of the D* Lite algorithm. The generation speeds at worse were merely just over half of the speeds that it took the D* Lite to regenerate based on the information that it had. Lastly, we noticed that when making use of the Theta* algorithm there was some instances where the agent would get caught out on a blocking tile on the map and was unable to move. After inspection, it appeared that it was to do with the way that the path was generated, in that the used line drawing algorithm did not detect an intersection at that given point and select a better parent.

C. *Zig-Zag*

The purpose of this test was to determine how accurate the path generation was a whether there was any difference at between the paths when traversing across a simple terrain. As expected, we noticed that the Theta* algorithm was capable of generating the sharpest path and as such was able to have the shortest traversal time overall. Furthermore, A* was capable of generating a viable path as well but not quite as fast in traversal as the Theta* of which is understandable.

Moreover we noticed that the performance of the D* Lite algorithm was questionable both in traversal and the generation of the path. Considering that the agent had no prior information regarding the turning points within the map, there would be slight confusion as it would tend to lean towards the left part of the maze when descending the level. We assume that this is simply because the distance heuristics between the agent and the cake is suggesting that is the most appropriate direction to head in until it learns the shape of the Zig-Zag. There were also some unfortunate instances in which the agent would go back on its path even though it did not necessarily go towards the cake but then quickly changed again once it had recognized the environment.

When we enabled D* Lite to also have full awareness of the environment by informing it of all the impassable nodes, it took approximately 3 seconds to generate an appropriate path to the cake. This is surprising considering the size of the environment but immediately we can deduct this as simply not being feasible for usage. We attempted to make use of the algorithm again but this time did not inform the pathfinding of any of the surrounding environment to the agent. Afterwards we began recording re-planning times between 0-15ms. Unfortunately the path traversal was significantly slower than the other methods.

Lastly, with the usage of the JPS A* pathfinding enhancement we noticed similar traversal speeds to that of Theta* start but taking substantially less amount of time to generate the path as a whole. This is displayed by times ranging from 0 to 15 ms of which match that of the re-planning times demonstrated by D* Lite.

FUTURE WORK

Most notably within the research we noticed that utilizing aesthetic path optimizing methods such as Theta* caused

problems due to the way that we were determining collisions between the parent of a node and one that is being observed in the search space. Unfortunately there isn't complete accuracy in determining the shortest path which then leads to our ants getting stuck into the terrain within some circumstances. We believe that this may simply be down to a granularity issue when calculating the increments and should this be the case then it's something we shall follow up later.

In comparison to our other pathfinding methods, we were simply able to check whether the next path node that the ant was traversing to was considered blocked and if so, then to simply re-plan. While Theta* is a good method for shortening path traversal times, based on our results we can conclude that it's not necessarily appropriate within an environment that alternates frequently with dynamically positioning objects. In the future, we could consider improving this problem by simply placing path nodes in between the points that are determined by the Theta* algorithm. This way we can check these nodes during the traversal of the path to determine whether or not there is a blockage from one of the moving boxes. This could then be achieved by re-using the line-drawing algorithm to determine if there are any dynamic boxes within the environment that intersect our ray-trace checks. If so, we determine the distance between the box and the ant and check accordingly. We faced a similar issue with JPS A* too in that because of the nature of the algorithm, there were no intermediate nodes that we could use to determine whether there was a blockage.

Considering the problems that we faced with the inaccuracies of the Theta* pathfinding method, it would be considered preferable in the future to make use of something such as the Jump-Point-Distance. The reason for this is that the path traversal time is roughly the same and computationally less expensive. Moreover, the algorithm would be much friendlier for platforms that suffer from strict memory constraints such as mobile devices. As another form of measurement diagnostics, we would like to pursue the idea of determining the memory consumption of each of these algorithms in total by determining the size of the path node objects stored in containers.

If we were to go ahead with some form of incremental heuristic pathfinding method to be used in our game *Ant Frenzy*, then we would consider enabling the sharing of the heuristic information that is generated when re-planning occurs in the environment. The reason for this is that we expect within the game to be a large quantity of ants that would play against the player. In order to minimalize the memory overhead from each entity that it is traversing the environment, it would be beneficial if this information could be shared amongst all the ants within the game state. An approach like this would require much more work and revision of the current code that we have for D* Lite. However considering the behavior of ants are typically similar to each other when moving as a cluster, we feel that something like this would be much more appropriate.

To conclude, we strongly believe that the usage of an algorithm is entirely down to the context that it's applied in. D* Lite demonstrated promising results considering the speed at which it could re-generate its path. Our concern however is more with the traversal between the current starting point and the goal. For the most part, it demonstrated that it was worse than our other algorithm. As such, considering that JPS A* demonstrated competitive generation speeds and had better traversal times suggests that perhaps incremental heuristic pathfinding is unnecessary. Ultimately we will put it down to how many entities are going to be in the environment at a given time. The likes of 30 ms path generation times displayed by JPS A* within a large environment will not display any performance loss whatsoever should it be a minimal amount of agents using it repeatedly in the game state.

REFERENCES

[1] X. Sun, W. Yeoh, and S. Koenig, "Moving target D* Lite," in Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1, Richland, SC, 2010, pp. 67–74.

[2] S. Koenig and M. Likhachev, "D*lite," in *Eighteenth national conference on Artificial intelligence*, Menlo Park, CA, USA, 2002, pp. 476–483.

[3] A. Nash, "Theta*: Any-Angle Path Planning for Smoother Trajectories in Continuous Environments," *AI Game Dev*, 08-Sep-2010. [Online]. Available: http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/. [Accessed: 31-Mar-2013].

[4] "Jump Point Search," *Shortest Path*. [Online]. Available: http://harablog.wordpress.com/2011/09/07/jump-point-search/. [Accessed: 10-Apr-2013].

[5] A. J. Champandard, "Near-Optimal Hierarchical Pathfinding (HPA*)," *AI Game Dev*, 11-Oct-2007. [Online]. Available: http://aigamedev.com/open/review/near-optimal-hierarchical-pathfinding/. [Accessed: 22-Apr-2013].

[6] A. J. Champandard, "Hierarchical Task Networks for MIssion Generation and Real-Time Behaviour," *AI Game Dev*, 20-Apr-2010. [Online]. Available: http://aigamedev.com/open/coverage/htn-planning-discussion/. [Accessed: 20-Apr-2013].

[7] T. Lindeijer, "Tiled Map Editor - Main page," *Tiled Map Editor*. [Online]. Available: http://mapeditor.org.

[8] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime Dynamic A*: An Anytime, Replanning Algorithm," presented at the Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 2005.

[9] P. Lester, "Using Binary Heaps in A* Pathfinding," *Using Binary Heaps in A* Pathfinding*, 11-Apr-2003. [Online]. Available: http://www.policyalmanac.org/games/binaryHeaps.htm.

[10] "EuclideanDistance - Wolfram Alpha," *Euclidean Distance - Wolfram Alpha*. [Online]. Available: http://reference.wolfram.com/mathematica/ref/EuclideanDistance.html.

[11] A. J. Champandard, "Pathfinding in Static and Dynamic Environments and the Future of Multi-core HPA*," *AI Game Dev*, 13-Oct-2010. [Online]. Available: http://aigamedev.com/open/review/hpa-future-multicore/. [Accessed: 22-Apr-2013].

[12] H. Tulleken, "Bézier Path Algorithms," *devmag.org.za*, 2011. [Online]. Available: http://devmag.org.za/2011/06/23/bzier-path-algorithms/. [Accessed: 11-Mar-2013].

[13] D. Harabor and A. Grastien, "Online Graph Pruning for Pathfinding on Grid Maps," *Association for the Advancement of Artificial Intelligence*, 2011.

[14] E. F. Krause, Taxicab Geometry: An Adventure in Non-Euclidean Geometry. Dover Publications, 1987.