2/25/2025

Believability: A Comparison of Pathfinding Algorithms

GDEV60001 Games Development Project

Harry Watts
W016075M

Supervisor: Davin Ward

Second Assessor: James Banton

Contents

Glossary 3
List of Figures 3
Abstract 6
Introduction 6
Aim and Objectives 7
Literature Review 8
How AI has Changed 8
Pathfinding 10
1. Dijkstra's Algorithm 10
2. A* Pathfinding 12
3. Dynamic Pathfinding 18
Planning 28
4. Finite State Machine 28
5. Behaviour Trees 28
6. Reactive Behaviour Trees and MARPO 30
7. Automated Planners 32
Research Methodologies 32
8. Artifact Creation 33
9. Testing and Results 33
10. Interpreting the Data 35
Results and Findings 35
11. Computational Results 35
12. Heatmap Results 41
13. Questionnaire Responses 48
Discussion and Analysis 52
14. Computational Discussion 52
15. Heatmap Discussion 53
16. Questionnaire Discussion 55
Conclusion 56
Recommendations 57
Appendix 59

Bibliography 60

Glossary

∈	Element of a set
⊆	Subset
A*	A-Star
Agent	AI Actor
AI	Artificial Intelligence
D*	Dynamic A*
FPS	Frames Per Seconds
FSM	Finite State Machine
GOAP	Goal Oriented Action Planning
Locally Consistent	Nodes g-values equal the rhs-values
LPA*	Lifelong Planning A*
MARPO	Movement, Avoidance, Routing, Planning and Orders
Nav Mesh	Navigation mesh
Rhs	Right Hand Sides
RTS	Real Time Strategy

List of Figures

Figure 1.1 Simple Binary Tree 11
Figure 1.2 Basic Search Space Graph containing several nodes. 12
Figure 2.1 $h*n = cost\pi*n, goal$ 13
Figure 2.2 $hn \leq h*n$ 13
Figure 2,3 $hn = xn - xfinal + yn - yfinal$ 13
Figure 2.4 $hn = xn - xfinal2 + yn - yfinal$ 14
Figure 2.5 $hn = maxxn - xfinal, yn - yfinal + 0.41 \cdot minxn - xfinal, yn - yfinal$ 14
Figure 2.6 $fx = gx + hx$ 14
Figure 2.7 Directed Weighted Graph 15
Figure 2.8 Grid with a starting node (in green) and the frontier nodes (in blue) 16
Figure 2.9 Pseudocode of A* 17
Figure 2.10 Dijkstra's algorithm vs A* created in Pathfinding.js (qiao, 2023) 18
Figure 3.1 Algorithm for PROCESS-STATE in D* algorithm (Stentz, 1994) 20

Figure 3.2 Algorithm for MODIFIY-COST (Stentz, 1994) 20
Figure 3.3 Average time to complete paths for A*, Dijkstra's algorithm and D* (Iskanda, et al., 2020) 21
Figure 3.4 comparing the time complexities of A* vs D* (Reeves, 2019) 22
Figure 3.6 $0 < cs,s' \leq \ \infty\#3.5$ 23
$g * s = \quad 0 \qquad if\ s = sstart, mins' \ \in predsg * s' + cs', s \qquad Otherwise$ 23
Figure 3.7 First and second pass pathfinding search (Koenig, et al., 2004) 24
Figure 3.8 $Consistent = gx == rhsx$ 24
Figure 3.9 $Over - Consistent = gx > rhsx$ 25
Figure 3.10 $Under - consistent = gx < rhsx$ 25
Figure 3.11 Graph showing possible route taken by AI using D* lite 25
Figure 3.12 Graph shows node X is now impassible 26
Figure 3.13 Updated graph with updated values 27
Figure 5.1 Basic Behaviour Tree Layout 29
Figure 6.1 Basic MARPO task layout 30
Figure 6.2 Order of priority in RTS games 31
Figure 9.1 Table of Quantitative vs Qualitative (Bell, et al., 2015) 34
Figure 11.1 Maze One Map 36
Figure 11.2 Maze One Results of Testing 37
Figure 11.3 Map of Maze Two 37
Figure 11.4 Maze Two Results of Testing 38
Figure 11.5 Map of Decadence from Hotline Miami (Dennaton Games, 2013) 39
Figure 11.6 Results of Testing Decadence 39
Figure 11.7 Map of No Talk from Hotline Miami (Dennaton Games, 2013) 39
Figure 11.8 Results of Testing No Talk 40
Figure 11.9 Map of Metro from Hotline Miami (Dennaton Games, 2013) 40
Figure 11.10 Results of Testing Metro 41
Figure 12.1 Decadence Heatmap with Dijkstra's Algorithm 42
Figure 12.2 Decadence Heatmap with A* 43
Figure 12.3 Decadence Heatmap With D* Lite 43
Figure 12.4 No Talk Heatmap with Dijkstra's Algorithm 44
Figure 12.5 No Talk Heatmap with A* 45
Figure 12.6 No Talk Heatmap with D* Lite 45
Figure 12.7 Motel Heatmap with Dijkstra's Algorithm 46
Figure 12.8 Metro Heatmap with A* 47
Figure 12.9 Metro Heatmap with D* Lite 47
Figure 13.1 Results of Question One of Questionnaire 48
Figure 13.2 Results of Question Two of Questionnaire 49
Figure 13.3 Results of Question Three of Questionnaire 50
Figure 13.4 Results of Ranking the paths 51

Figure 13.5 Breakdown of Results of Ranking the paths 51
Figure 14.1 Maze Two Paths A*(Left) and D* Lite(Right) 53

Abstract

In most games an Artificial Intelligent agent needs to be able to navigate a space quickly and efficiently. To this end pathfinding has been driven to be increasingly more efficient from the adaption of Dijkstra's algorithm to the globally used A* and not just in games robotics and GPS navigation have all adapted the use of pathfinding. However, unlike robotics and real-world navigation, agents in games need to appear intelligent to the player meaning the believability of the path taken needs to be considered. This must mean that for a path to be considered acceptable the algorithm must satisfy two criteria: efficient and believable. An investigation into several pathfinding algorithms were tested for their believability and efficiency but also what a player deems to be believable, this included getting players to navigate the same maps to compare to the pathfinding algorithms. To ensure a range of algorithms were tested both static and dynamic algorithms were included those being: Dijkstra's Algorithm, A* and D* Lite with the results showing that despite the enhancements made to the algorithms the environment has as much of an effect on the algorithm as the efficiency of it.

Introduction

The nature of game Artificial Intelligence (AI) is to find that sweet spot of intelligent behaviour, believability and fun for the player, creating AI agents that are too smart or challenging for the player can disrupt the experience of the game while creating AI that cannot react to obstructions or player actions can also ruin immersion. Many games today use the A* pathfinding algorithm to navigate an environment, due to its effectiveness at navigating around static obstacles. However, as A* calculates a path based upon the environment at the time, if presented with a dynamic obstacle it's possible the AI will become stuck and without any way to react to it or break out of the path its currently following. This could cause an agent to constantly walk into an obstacle indefinitely ruining the believability of that agent. But navigation is only one half of the reactive coin, should an obstacle appear it is one thing to recalculate the path around it but a completely different problem if the player has locked a door and the only way through is to find the key that opens it.

There are a few pathfinding algorithms that have been developed in order to tackle this problem, the likes of D*, Dynamic A*, LPA*, Lifelong Planning A* and D* lite, have all achieved this in some way by recalculating parts of the path in front of the agent when the path has become blocked rather than starting again from the agents position to the goal. This iteration over the current viable path prevents unwanted computation time of a complete recalculation which when expanded on too many agents in a game can take time away from other key areas such as physics calculations for graphic rendering (Roberts, 2022).

To make an AI feel more life-like and believable then the agent needs some way to react to the goings on in the world, planning a path is one thing but if the agent tries to travel through a door that is locked by the player without something stopping the AI following the current path the agent will continue to try and walk through the door. This means that the agent needs a way of not just planning but also to react to the new scenario in front of it, for this reason more advanced ways of decision making than Finite State Machines (FSM) need to be utilised where the agent will be allowed to break out of the current task and follow the new higher priority task. This also means that a series of goals need to be established so that the tasks can be order based upon the goals of the agent.

Aim and Objectives

AI agents can navigate a space in many ways, from the simple and static methods to the more complex dynamic and reactive approaches that will allow the agent to adapt to an environment prone to changing. The goal will be to determine how an AI agent can navigate a space dynamically and efficiently with considerations of believability and computational outcomes.

- Multiple areas of pathfinding and decision making will be explored including their origins to ensure all understanding and conclusions are based upon correct, current and fundamental knowledge.
- How player interact and immerse themselves in games will also be researched to help formulate the methods in which the research done can be tested against. This will include understanding how a player will move through the same space the agent does to compare their routes.
- A simulation will be built using the information gained to determine a believability vs efficiency cost comparison and determine which method proves more cost effective.

To ensure all aspects are overviewed sufficiently future work and areas for potential development will be outlined and areas that might offer improvements will be considered.

Literature Review

How AI has Changed

Since some of the first video games created, AI has played a major role in providing immersion and challenge for the player (Dill, 2013). From the simple two state systems used in Pac-Man (Namco, 1980) to the highly complex AI Director and Behaviour Tree's utilised in Alien: Isolation (Creative Assembly, 2014) AI in games has always aimed to engage the player.

AI in games has been used for multitude of purposes from creating fun and engaging enemy agents that act as a barrier for the players progression to companions designed to assist the player or provide some story elements (Dyckhoff, 2017; Millington, 2019). Most agree that the underlining principle to create compelling and engaging AI agents is the illusion of intelligence, taking an object in the game and bringing it to a semblance of life in the eyes of the player can make or break a games AI (Rabin, 2017; Roberts, 2022).

Creating the illusion of intelligence is a delicate balance, develop an agent that is it too intelligent and the game might engage the player into new strategies, or become too challenging for the player to enjoy. Opposingly designing a dumb agent can add personality to a game or break the immersion and ruin the experience (Roberts, 2022; Bourg & Seamann, 2004). Preserving a games immersion is key in allowing the player to experience the flow state. The Flow state or Flow is a psychological effect that describes a pleasurable immersion experienced by a player (Soutter & Hitchens, 2016).

The eight components of Flow

- Clear Goals
- High Degree of Concentration
- Loss of Self-reflection
- Time Distortion
- Direct and Immediate Feedback
- Sense of Personal Control
- Intrinsically Rewarding
- Balance between Ability Level and Challenge

It is clear from this list that Flow is an intrinsic value baked into many games, challenging fights, quest rewards and direct personal control makes it easier for games to maintain the flow state for long periods of time (Soutter & Hitchens, 2016). However, this benefit also means that once broken it can prove difficult to restore.

This pretence of sentience has permeated games since players were shot at by aliens in Space Invaders (Tatio, 1978) while the AI design was simple it demonstrated intelligence through seemingly coordinated attacks. The enemy ships moved and acted together always advancing on the player, seemingly working to a strategy and together while not having the capability to do so (Rabin, 2017). This simple yet effective design exploded the popularity of AI in games but also in complexity allowing an AI agent to navigate an environment using pathfinding techniques, or chase down a player using steering behaviours as seen in Doom (id Software, 1993) and The Elder Scrolls II: Daggerfall (Bethesda Softworks, 1996). This adaption of pathfinding algorithms changed how the player interacted with the game, creating unique combat encounters that challenge the player rather than memorising patterns (Thompson, 2024).

Creating agents that can challenge the player through seemingly intelligent tactics was not the only avenue of adaptability explored for AI, both Left 4 Dead (Valve South, 2008) and Alien: Isolation (Creative Assembly, 2014) utilise an AI director that controls how the AI will react but both use this director in very different ways. Left 4 Dead keeps track of how the game is currently progressing based upon preplanned parameters such as the players progression speed, the quicker they progress the more enemies are spawned and if the players seem to be having difficulties progressing less enemies or less special enemies maybe spawned (Thompson, 2024). Alien: Isolation uses the director method in a much more targeted way, rather than having the director control numerous enemies at any point, it oversees one agent. Using a two-pronged approach, one is the agents AI brain containing a Behaviour Tree for the agent including, but not limited to, pathfinding and how the alien might search a room. Meanwhile the director has full knowledge of the current game state, always knowing where the player is and feeding the alien information based upon if the player has gone too long without an encounter it keeps track using a menace meter, should this menace meter become full then the director will send the player to a different area giving the player a brief respite (Thompson, 2017). This ability to adapt to the player's actions gives the sense that the AI is a living entity that not only has intelligence but can learn how to beat the player, improving the immersion and fun of the game without creating an AI that is unbeatable (Rabin, 2017).

Pathfinding

Pathfinding or Path Planning is a commonly used method of navigating, a practice that can be observed in the real world. Humans will find paths based upon what they know about their environment and maps they have created in memory (Rahmani & Pelechano, 2021). Computer games operate differently, without the human aspect of navigation such as memory, orientation and personal bias (Rahmani & Pelechano, 2021), AI takes what it knows to be fact about its environment, this can be checkpoints or a fixed path that is set by the designer, or information contained in the navigation mesh that could include what is traversable and what is not (Millington, 2019).

Spatial Representations

To give an agent the ability to see the environment must first be split into navigational data that informs the agent of its surroundings (Green, 2024). The simplest method is to divide the search space into a grid of nodes that hold whether they are traversable or not (Roberts, 2022). Turning the environment into a grid works well for 2D environments however, 3D environments more information may be needed to accommodate the additional dimension, i.e. jumping or crouching (van Toll, et al., 2016). Nav Meshes can bridge this gap by converting the environment into cubes in a process called voxelisation and assessing areas that are traversable in the space (Brodén & Bohlin, 2017).

1. Dijkstra’s Algorithm

In 1959 a Computer Scientist by the name Edsger W. Dijkstra aimed to solve the shortest path theory when attempting to navigate a graph or tree by traversing between the nodes (Millington, 2019). Dijkstra took the understanding that a tree is built by creating a path from a parent node to its child when instantiated, this limitation of one path between two nodes allows the navigation from any node on the tree to the other (Dijkstra, 1959). Figure 1.1 shows a simple binary, to navigate the tree the search there are two basic search algorithms Depth first and Breath first. Both searches will start at the root node(A) and if Depth first will travel to the lowest node, it can on the left side before moving to the right making the order of nodes searched A, B, C, E, F, D, G. Breath first travels across from left to right before exploring further down the tree, this would make the order of the search A, B, C, D, E, F, G. Dijkstra algorithm looks to navigate the tree differently, by utilising the single path between each node traversal from node G to node C moves along two nodes to arrive at the destination (Dijkstra, 1959). It is important to note that using Dijkstra’s algorithm from G to C the path from B to A will also be searched as it must be confirmed to not be C before is discounted.

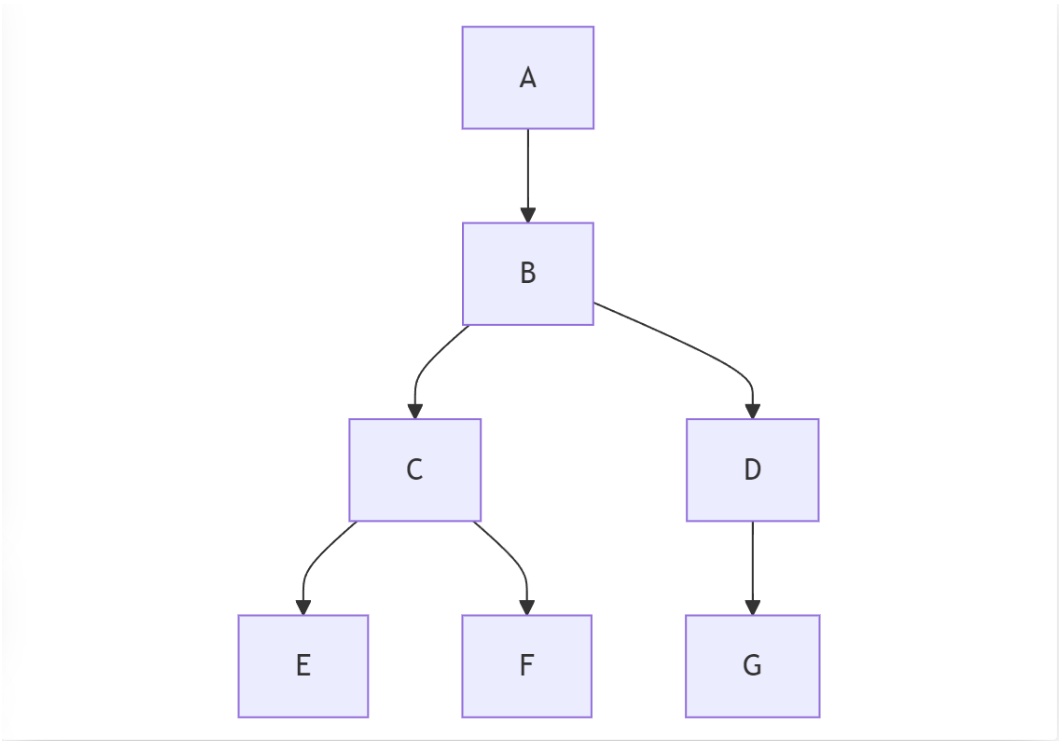


Figure 1.1 Simple Binary Tree

While Dijkstra's algorithm works for binary trees its real potential is shown in navigating graphs Figure 1.2 shows a very simple search space graph structure following the rules laid out by Dijkstra, each node must have one and only one connection to each other node (Dijkstra, 1959). To navigate from node A to node D in figure two there are two possible routes to take A->B->D or A->D while it is easy to visualise the shortest path as being straight from A to D computers cannot differentiate between the two Dijkstra's Algorithm solves this by evaluating the cost of the path taken taking the shorter one and deleting the longer ones (Millington, 2019). AI in games use Dijkstra's shortest path algorithm by treating the environment as a graph, this can be done by splitting the game space into a perfect graph, creating waypoints or using a nav mesh, to demonstrate this take Figure 1.2 and imagine it describing key points on a map, F might be inside a building or A might be a bridge, if the AI agent is currently standing on node E how would it get to node G? Dijkstra's Algorithm provides a way of navigating the world and will supply the shortest path between the two nodes, however, is not without drawbacks (Roberts, 2022).

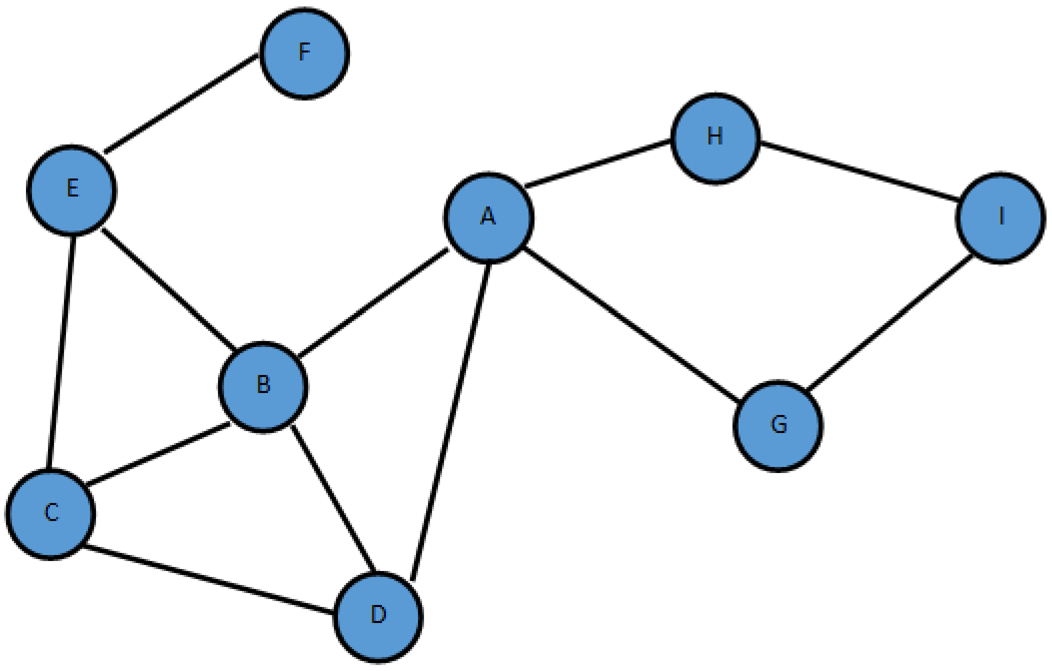


Figure 1.2 Basic Search Space Graph containing several nodes.

The first and biggest drawback of Dijkstra's algorithm is also its biggest attraction, it will find the shortest path no matter what this includes checking in all directions until the path is found (Roberts, 2022). A path from B to H will need to visit nodes C, D, and E regardless of the direction of H, once the path is found the incorrect paths will be discarded but after they have used up processing resources (Millington, 2019). This method of searching is known as an Undirected Search, there is no choice in which direction the algorithm will search in and has the potential of searching the entire tree before finding the path (Roberts, 2022).

2. A* Pathfinding

Dijkstra's algorithm used a mathematical approach to finding the shortest path, its aim is to find the shortest path no matter the cost, A* (A-star) built upon this by combining this method with a heuristic approach, this, unlike Dijkstra's algorithm, allows A* to be a directed search approach giving the algorithm a general direction to travel in (Hart, et al., 1968; Roberts, 2022).

Heuristics

A* being able to direct its search limits the paths searched, increasing its efficiency while searching the games environment for the fastest path. Unless the path is a not viable, A* does not discard any nodes, using a best first search it searches through the nodes that take the path closer to the end (Liu, 2023). To aim the search in the direction of the goal A* uses Heuristics *(Greek: 'to find')* to guide which node is best by estimating the cost from each node to the end when calculating the cost of the next step. If the cost is the lowest in the nodes around the current point, then this path is searched first as the 'best' as it is moving in the right direction (Airlangga, 2024).

The type of heuristic used can affect the outcome of the path based upon the answer returned (Roberts, 2022). Changing the games environmental layout to match the heuristic may provide a better outcome; Manhattan tends to favour four neighbour nodes whereas Octile operates better with eight neighbours (Rivera, et al., 2020).

Heuristics tend to fall into two categories perfect and admissible. Perfect heuristics, denoted *h**, dictate that the value of each node on the path is the true cost-to-go expressed in figure 2.1 (Kirilenko, et al., 2023).

$$h * (n) = cost\left(\pi * (n, goal)\right) \tag{2.1}$$

The heuristic is called admissible if it never overestimates the true go-to-cost for each node, expressed in figure 2.2, this style of heuristic in use with the A* algorithm is widely accepted to find the best solution to a closed grid (Kirilenko, et al., 2023).

$$h(n) \leq h * (n) \tag{2.2}$$

Manhattan Heuristic

The Manhattan approach is one of the most common heuristics used in pathfinding, taking its name from the Manhattan borough of New York. Manhattan imagines the environment as city blocks taking the absolute distances in the axes to estimate the distance needed to arrive at the end point (Roberts, 2022; Guo & Luo, 2018). Manhattan Distance expression shown in figure 2.3.

$$h(n) = \left|x_n - x_{final}\right| + \left|y_n - y_{final}\right| \tag{2.3}$$

Euclidean Heuristic

Euclidean distance uses trigonometry to calculate the exact distance between the start and end points. The distance estimate being the hypotenuse of the triangle created by using Pythagoras' theorem on the x and y distances from start to end position (Roberts, 2022; Guo & Luo, 2018). Euclidean Distance expression shown in figure 2.4.

$$h(n) = \sqrt{(x_n - x_{final})^2 + (y_n - y_{final})^2} \qquad 2.4$$

Octile Heuristic

Octile distance is a variation on the Manhattan heuristic it takes the same distances but multiplies the shortest distance by 0.41 adding it to the longest. This heuristic is best used on a grid and is the most accurate on one as Manhattan will tend to overestimate and Euclidean will underestimate distances. Assuming that diagonal and orthogonal movement is allow the angles of movement align with the 45- or 90-degree angles that octile prefers (Rabin & Sturtevant, 2013). Octile Distance expression shown in figure 2.5.

$$h(n) = max\left((x_n - x_{final}), (y_n - y_{final})\right) + 0.41 \cdot min\left((x_n - x_{final}), (y_n - y_{final})\right) \qquad 2.5$$

The A Algorithm*

Although the A* algorithm did not originate in the games industry it has found its place as the most popular when it comes to calculating a path with the real-time demands of games (Liu, 2023). A*'s popularity has also expanded in the world of robotics (Lim, 1968), to Evacuation planning (Ji & Gao, 2007), A* has been implemented and adapted to fit a multitude of scenarios.

The equation used to calculate the total cost of the A* algorithm is shown in figure 1.6, where *f* denote the final cost of the node calculated by taking the computed cost so far, *g(x),* added to the heuristic guess form the node to the end, *h(x)* (Roberts, 2022).

$$f(x) = g(x) + h(x) \qquad 2.6$$

Directed Weighted Graph

As with Dijkstra's algorithm the environment space needs to be represented as a graph-like structure (Schoener, 2024), Figure 2.7 shows a modified version of the search space graph in figure 1.2, called a directed weighted graph where the cost between each node has been added. Still thinking of the graph as a game map the numbers between the nodes can be increased to show the cost to move between the two, in game this can represent different terrains i.e. if the path goes through water, it would be more difficult to navigate then the road (Bourg & Seamann, 2004).

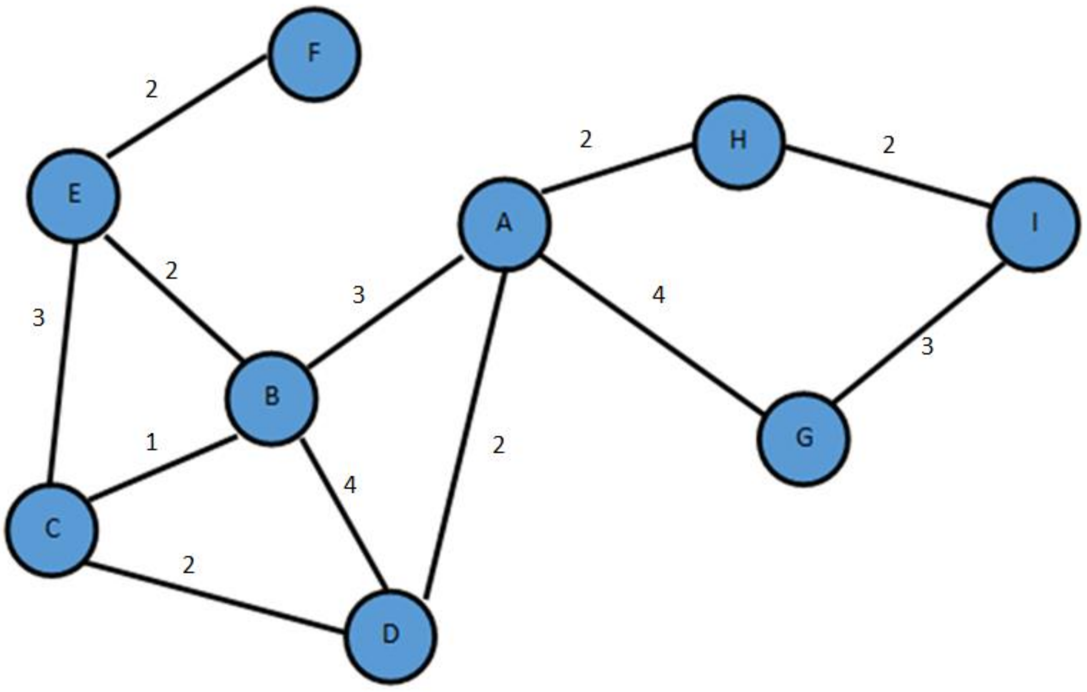


Figure 2.7 Directed Weighted Graph

To navigate between node A and node E there are multiple routes that can be taken, notably A->B->E is the shortest in number of nodes traversed and the cost of moving between each node. The directed weighted graph can highlight the advantages of using heuristics A* will first look through the cheap nodes and adding them to the open list. In this case the cheapest nodes to travel to are D and H both having the travel cost of 2. It is at this point the heuristic calculation will inform the pathfinding that its heading in the wrong direction, thus, redirecting the search to B and closer to the goal (Rafiq, et al., 2020).

Open and Closed Lists

To allow A* to search the grid, it needs to have some way to 'look around' and then evaluate the results. A* and Dijkstra builds up information about its environment using open and closed lists, the open list contains the frontier, nodes which have yet to be explored but are known, initially the closest to the current node, and the closed list, nodes that have already been explored (Lester, 2005). Figure 2.8 shows a simple grid the green square at the centre indicates the current or starting node and the blue squares, containing the numbers, immediately surrounding it are the frontier, and finally the red showing the goal.

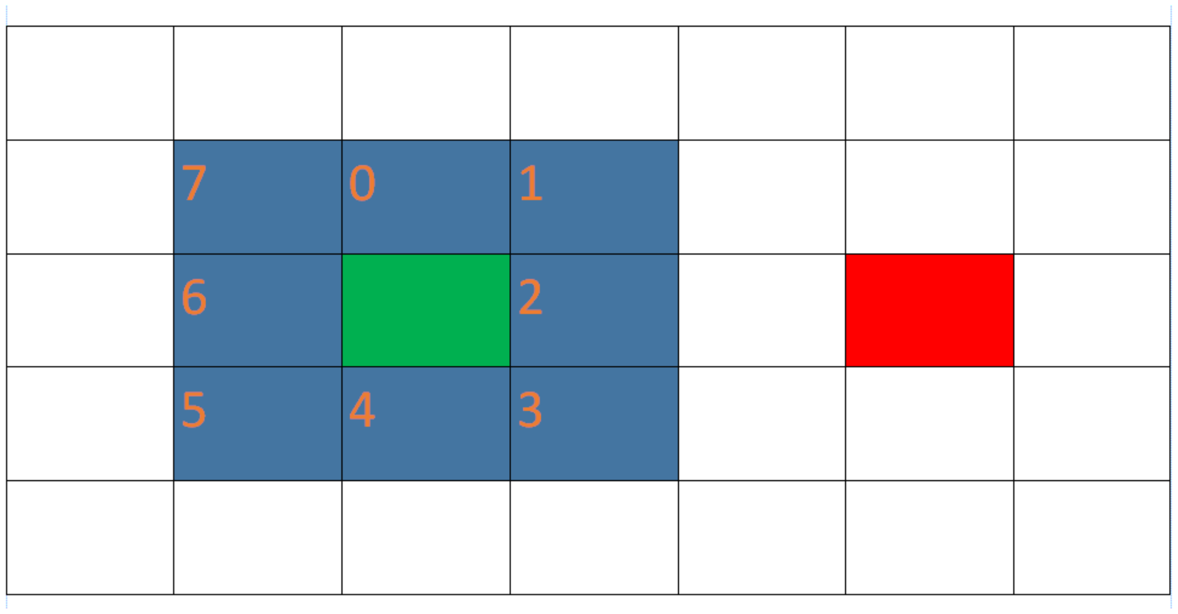


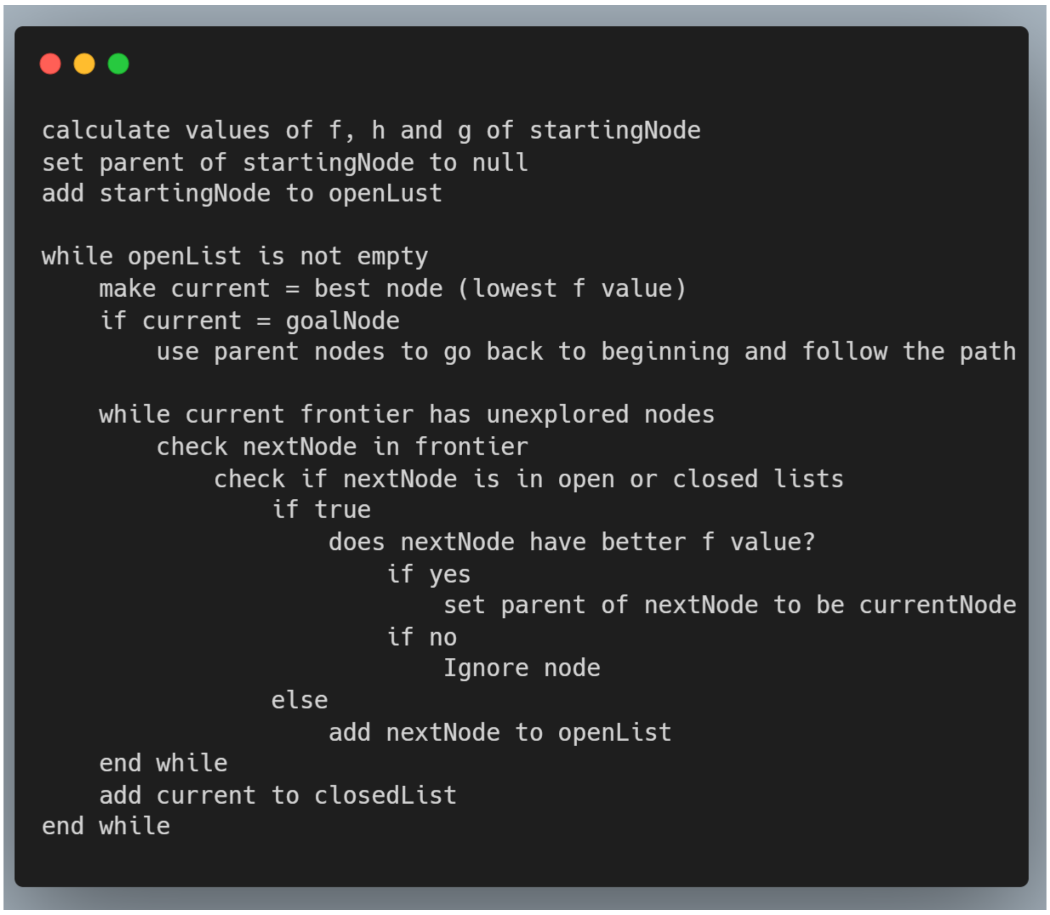
Figure 2.8 Grid with a starting node (in green) and the frontier nodes (in blue)

1. The start node is switched to the closed list, as it has now been explored.
2. The frontier is expanded, for this example the first node explored will be the square marked 0, right above the start node.
3. The frontier around this node is then explored, any nodes found not already on the open list are added with the current node as the parent and any on the closed list or unwalkable, i.e. walls, are ignored.
4. The nodes that are already on the open list are checked against the current node to determine which is better, comparing node 1 to the current node 0, node 1 is

closer and is cheaper to move straight there rather than going up to 0 then across so node 1 becomes the current node.

5. The new currents frontier is explored.
6. Repeat steps 3, 4 and 5 until the goal is found.

Figure 2.9 outlines this process in pseudocode (Roberts, 2022; Cui & Shi, 2010; Lester, 2005).

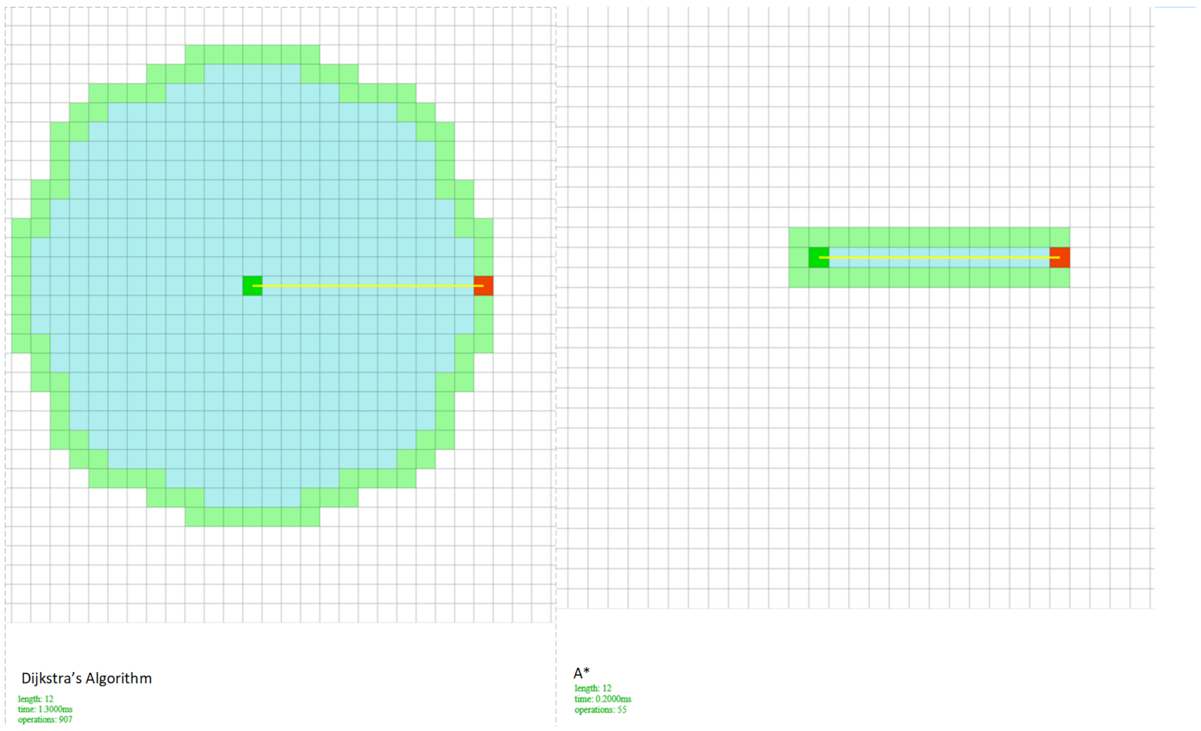


```
calculate values of f, h and g of startingNode
set parent of startingNode to null
add startingNode to openLust

while openList is not empty
    make current = best node (lowest f value)
    if current = goalNode
        use parent nodes to go back to beginning and follow the path

    while current frontier has unexplored nodes
        check nextNode in frontier
            check if nextNode is in open or closed lists
                if true
                    does nextNode have better f value?
                        if yes
                            set parent of nextNode to be currentNode
                        if no
                            Ignore node
                else
                    add nextNode to openList
    end while
    add current to closedList
end while
```

*Figure 2.9 Pseudocode of A**

*Dijkstra vs A**

Understanding how A* utilises heuristics the comparison between A* and Dijkstra's algorithm becomes more understandable. The inability to limit the direction of searching using heuristics leads to a lot more nodes explored, and calculations done. Figure 2.10 shows an example of Dijkstra's algorithm and A* looking for the same goal, A* using Manhattan heuristics, created using Pathfinding.js (Xu, 2023).



Dijkstra's Algorithm
length: 12
time: 1.3000ms
operations: 907

A*
length: 12
time: 0.2000ms
operations: 55

Figure 2.10 Dijkstra's algorithm vs A created in Pathfinding.js[1] (qiao, 2023)*

While the path is simple the difference between the two show the major advantages of A* and with games aiming for 60 fps (Frames Per Second) (Madhav, 2018) saving 1.1ms per path calculation can really add up.

3. Dynamic Pathfinding

The consensus deems A* as the backbone of pathfinding, with many focusing on how to optimise A* over reinventing the wheel (Bourg & Seamann, 2004). One avenue of optimisation is adapting A* to work in a dynamic environment, as explored standard A* is an improvement upon Dijkstra's, but they are both limited to static environments, using A* in a changeable environment can lead to agents becoming blocked or stuck with the only solution being recalculating the entire path from the new position (Mäntysalo, 2024).

In a best-case scenario, the entire environment is known and static meaning the path only needs to be calculated once, for this situation A* works well. However, in the world of exploration robotics the entire environment cannot be fully known, and obstacles encountered make the current path invalid (Stentz, 1994). This forces the path to be recalculated leading to inefficiencies that can grow exponentially depending on the

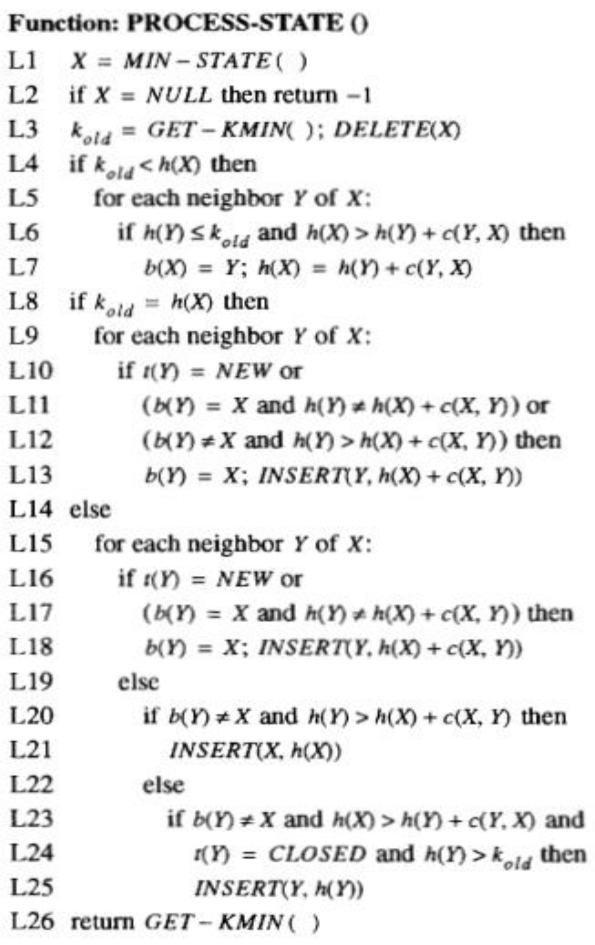
[1] Pathfinding.js [https://qiao.github.io/PathFinding.js/visual/]

number of obstacles encountered and amount of information about the area to begin with.

D Algorithm*

In 1994 research professor Anthony Stentz proposed a solution to dynamic or unknown environments, using their role in the field of robotics, A* cannot adapt on the go, once the path is formed new information will not alter the current path until the robot became stuck and a new path with the new information could be calculated. A dynamic solution was proposed based upon the A* algorithm but rather than being set once the solution would recalculate the problem nodes, that have changed, as new information is collected rather than recalculating the entire path, thus gaining its name D* or Dynamic A* (Stentz, 1994).

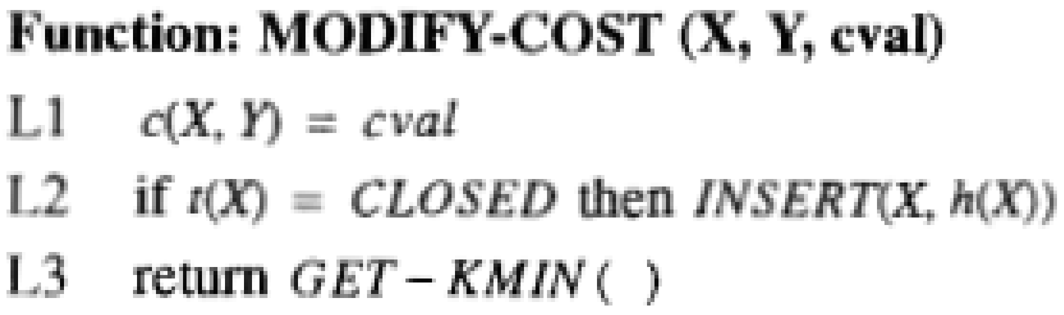
D* uses two main methods *PROCESS-STATE,* calculates the best path cost to the destination and *MODIFIY-COST,* used to change the cost of the current arc and reinsert the node onto the open list (Stentz, 1994). Unlike A*, D* works backwards from the destination to the start point, *PROCESS-STATE* is constantly called until the start is found and using the back-pointers of each node travelled through creates a path, this is very similar to the method used by A* to find the path. The algorithm for *PROCESS-STATE* can be found in figure 3.1.



```
Function: PROCESS-STATE ()
$X = MIN-STATE()$
if $X = NULL$ then return $-1$
$k_{old} = GET-KMIN()$; $DELETE(X)$
if $k_{old} < h(X)$ then
  for each neighbor $Y$ of $X$:
    if $h(Y) \leq k_{old}$ and $h(X) > h(Y) + c(Y, X)$ then
      $b(X) = Y$; $h(X) = h(Y) + c(Y, X)$
if $k_{old} = h(X)$ then
  for each neighbor $Y$ of $X$:
    if $t(Y) = NEW$ or
      $(b(Y) = X$ and $h(Y) \neq h(X) + c(X, Y))$ or
      $(b(Y) \neq X$ and $h(Y) > h(X) + c(X, Y))$ then
      $b(Y) = X$; $INSERT(Y, h(X) + c(X, Y))$
else
  for each neighbor $Y$ of $X$:
    if $t(Y) = NEW$ or
      $(b(Y) = X$ and $h(Y) \neq h(X) + c(X, Y))$ then
      $b(Y) = X$; $INSERT(Y, h(X) + c(X, Y))$
    else
      if $b(Y) \neq X$ and $h(Y) > h(X) + c(X, Y)$ then
        $INSERT(X, h(X))$
      else
        if $b(Y) \neq X$ and $h(X) > h(Y) + c(Y, X)$ and
          $t(Y) = CLOSED$ and $h(Y) > k_{old}$ then
          $INSERT(Y, h(Y))$
return $GET-KMIN()$
```

Figure 3.1 Algorithm for PROCESS-STATE in D algorithm (Stentz, 1994)*

In *MODIFIY-COST*, shown in figure 3.2, the cost of traveling to the node is changed thus making the path incorrect adding X back into the open list to be re-explored in the *PROCESS-STATE* function, the new heuristic cost is then calculated for Y then adding Y to the open list allows the cost to be pass onto the child nodes of Y effectively 'fixing' the path with the new information (Stentz, 1994).

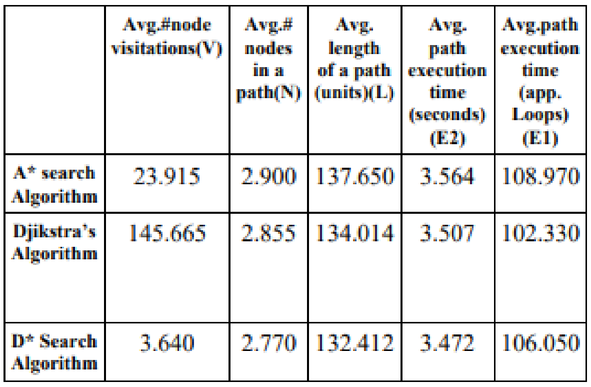


```
Function: MODIFY-COST (X, Y, cval)
$c(X, Y) = cval$
if $t(X) = CLOSED$ then $INSERT(X, h(X))$
return $GET-KMIN()$
```

Figure 3.2 Algorithm for MODIFIY-COST (Stentz, 1994)

While D* is similar to the A* in how it approaches finding the path D* drastically changes the primary function of A*, the shortest path is no longer the definitive answer to the problem rather D* looks for all paths to the goal, this allows paths to me modified rather than calculated in the moment an obstruction is encountered (Murphy, 2000). D* is known as a continuous re-planner whenever an obstacle is detected the path will be recomputed, in robotics this issue can present as a phantom reading, the obstacle does not actually exist but due to an instrument misreading the surroundings, potentially making the path recompute twice, first due to the obstacle and second due to no obstacle (Murphy, 2000). D* also uses two extra identifiers for when the path needs to change, RAISE, for when a node is more expensive then is was last time it was in the open list, and LOWER, for when a node is cheaper than is was last time, limiting the number of nodes that need to be rechecked (Reeves, 2019).

Figure 3.3 shows a table of results from a study done using pathfinding to solve a maze (Barnouti, et al., 2016), it shows that in this case D* visits significantly less nodes when finding the path and the execution time is on average faster than that of A*.



	Avg.#node visitations(V)	Avg.# nodes in a path(N)	Avg. length of a path (units)(L)	Avg. path execution time (seconds) (E2)	Avg.path execution time (app. Loops) (E1)
A* search Algorithm	23.915	2.900	137.650	3.564	108.970
Djikstra's Algorithm	145.665	2.855	134.014	3.507	102.330
D* Search Algorithm	3.640	2.770	132.412	3.472	106.050

Figure 3.3 Average time to complete paths for A, Dijkstra's algorithm and D* (Iskanda, et al., 2020)*

While Figure 3.3 shows that D* tends to be more efficient than A* it's important to note this is just one study figure 3.4 demonstrates the complexity of both algorithms responding to several obstacles. Increasing the number of obstacles in the scene the increases the complexity of A* becomes but decreases the complexity for D*. The use cases of each algorithm are dependent on the desired outcome a more static environment will hinder the advantages of using D* but make A* more attractive.

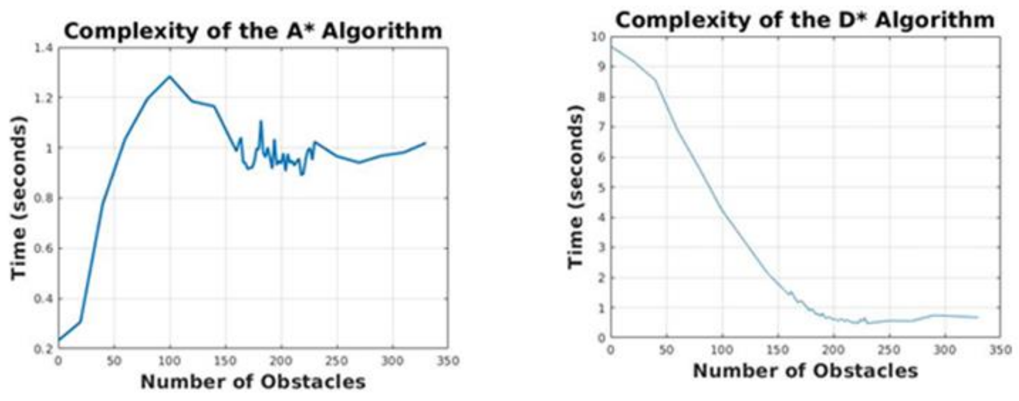


Figure 3.4 comparing the time complexities of A vs D* (Reeves, 2019)*

*Lifelong Planning A**

Created to solve the dynamic problem with A* for real world robotic applications makes D* difficult to justify in a game environment, mainly the first step of the algorithm is to calculate all possible paths to the destination this level of computation can be completed before the robot sets off. However in games each frame be completed in 16ms, while there are potential ways to navigate this issue, computing the paths over multiple frames, this will make the AI feel slow to react or the frames to drop both creating negative results (Murphy, 2000). Lifelong Planning A* (LPA*) limits the amount of computation by using heuristics to guide the shortest path and only recalculate the path on nodes that have changed, but also using the A* method once the shortest path has been found stop searching, this includes changed nodes (Koenig, et al., 2004).

LPA* uses the same method utilized in DynamicSWSF-FP by taking each nodes g-value, from A*, and rhs (right-hand sides) value, which informs the look ahead value by taking the smallest g-value of its neighbours plus the distance to get there, when these values match the node is called locally consistent (Koenig, et al., 2004). To know a path has become blocked the node is checked to be locally consistent, this can be done by checking if the g-value matches the rhs-value, if an inconsistency is detected the pathways of that node is recalculated to match again (Koenig, et al., 2004).

Mathematical Representations

To fully understand how the LPA* algorithm calculates its path is important to understand the maths behind the starting distances. Letting S represent the entire graph with finite vertices, then $\mathrm{succ}(s) \subseteq S$ shows the set of successors of $s \in S$ (s 'element of' S), equally $\mathrm{pred}(s) \subseteq S$ shows the predecessors. Figure 3.5 shows the cost of moving from s to $s' \in \mathrm{succ}(s)$ (Koenig, et al., 2004).

$$0 < c(s, s') \leq \infty \tag{3.5}$$

Figure 3.6 shows the start distance equation from s_{start} to s represented by g*(s). essentially stating that any node that is not the start must have a starting distance that satisfies the equation, being the minimum cost between current and next node (Zarembo & Kodors, 2013).

$$g*(s) = \begin{cases} 0 & if\ s = s_{start}, \\ min_{s' \in pred(s)}\left(g*(s') + c(s', s)\right) & Otherwise \end{cases} \tag{3.6}$$

Reducing the number of nodes that need to be searched lowers the demand of pathfinding, in games where most of the calculation time may be needed elsewhere it is crucial to be as efficient as possible. LPA* does this two ways the first incrementing over what the original pass searched, meaning every node in the game space is not searched again, the second is to only expand upon nodes that are now locally inconsistent reducing the number of changes to the nodes that have been altered (Koenig & Likhachev, 2001).

Figure 3.7 shows the number of nodes expanded on a first pass and if the environment changes respectively, in which its clear just bey the visual representation of the nodes that should the environment change the LPA* is more effective.

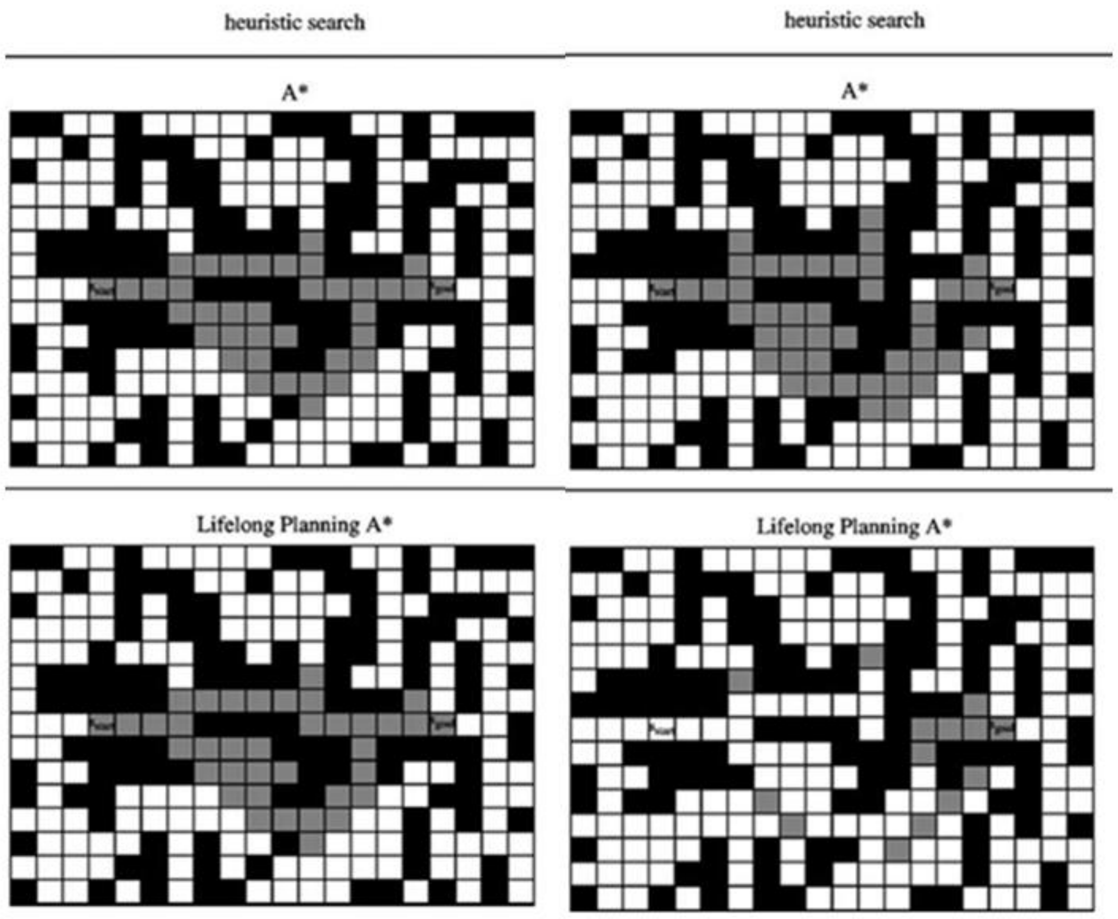


Figure 3.7 First and second pass pathfinding search (Koenig, et al., 2004)

D Lite*

D* lite was conceived as an improvement upon the LPA* algorithm but rather than searching from the start node D* lite begins at the goal node and works backwards, this makes the g-values goal distances rather than start distances (Koenig & Likhachev, 2002). Due to the incremental nature of the search the root node must remain the same, the goal being a fixed point makes it the perfect for this over the ever-changing position of the agent (Uzoeghelu, 2021).

Just as LPA* requires node to be locally consistent D* lite nodes can have one of three statuses Consistent, Over-consistent and Under-Consistent based upon them satisfying figure 3.8, 3.9 and 3.10 respectfully.

$$Consistent = g(x) == rhs(x) \qquad 3.8$$

$$Over - Consistent = g(x) > rhs(x) \qquad 3.9$$

$$Under - consistent = g(x) < rhs(x) \qquad 3.10$$

If any other than consistent is true the node is re-explored and updated with the new values (Uzoeghelu, 2021).

Figure 3.11 shows a very basic graph with a path an agent might take using D* lite. Firstly, it is key to remember that the goal note will have the g and rhs values of zero, due to D* lite's backwards search method where the path will propagate from the goal to the start. Second each diagonal step has been given a cost of 1.4 making it slightly easier to travel diagonally then travel only orthogonally. The arrows denote the path computed by the algorithm and if the game environment stays would also be the one followed.

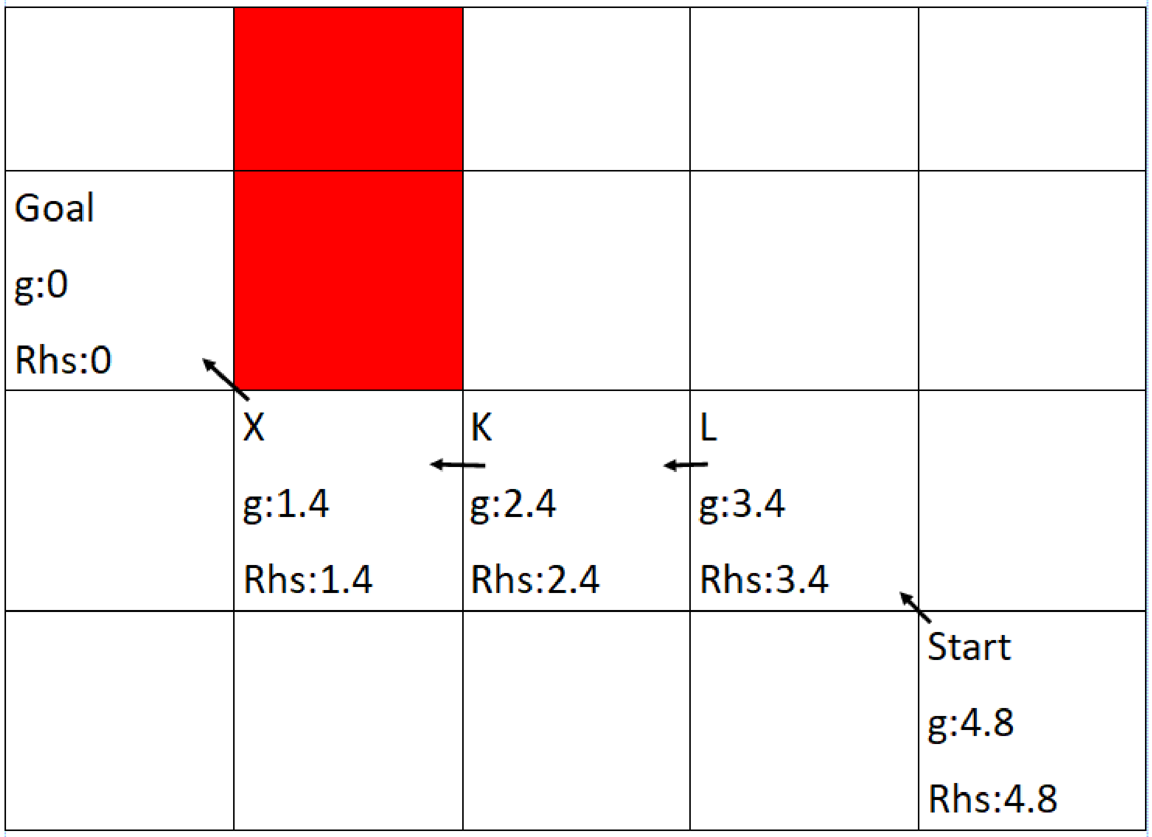


Figure 3.11 Graph showing possible route taken by AI using D lite*

However, if the agent gets to node K to find X is non-negotiable, shown in Figure 3.12, the value is updated making it locally inconsistent and added back onto the open list. From the open list the walkable neighbours are explored, recalculated and a new path found based upon them, shown in Figure 3.13 (Koenig & Likhachev, 2002).

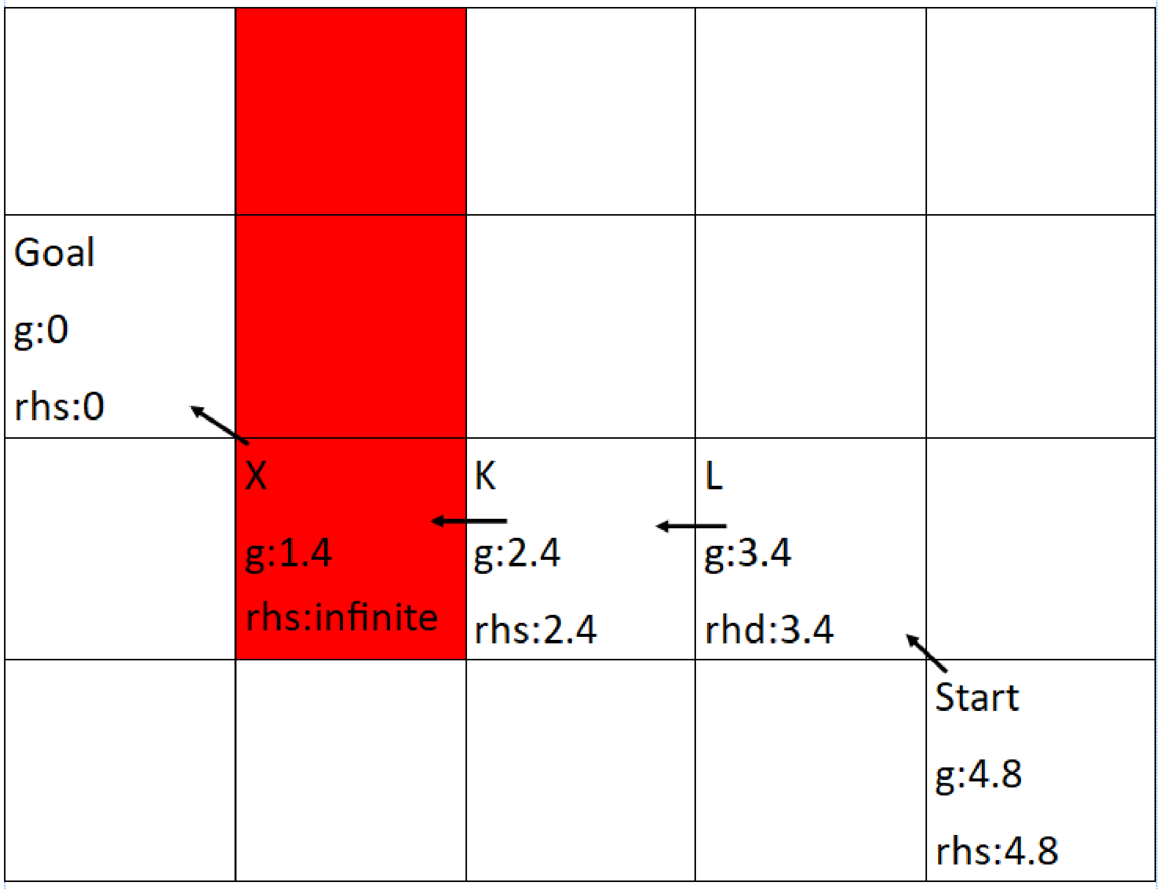


Figure 3.12 Graph shows node X is now impassible

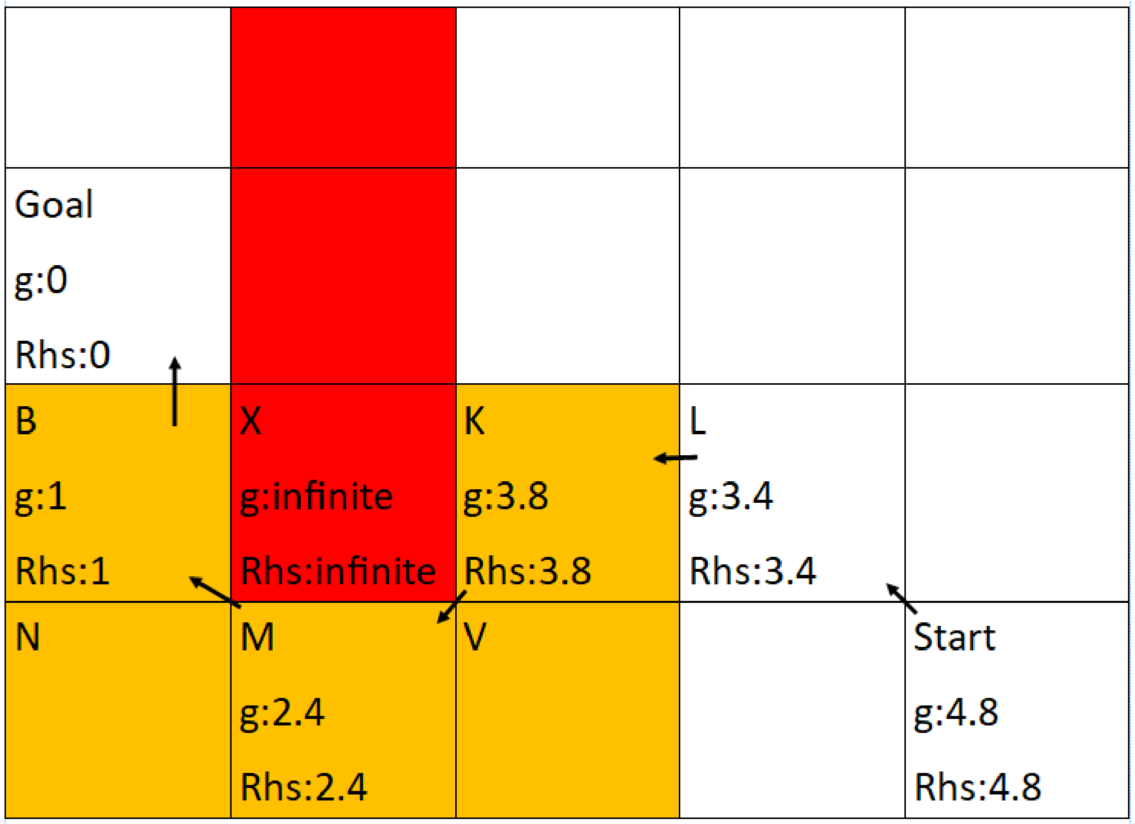


Figure 3.13 Updated graph with updated values

The nodes coloured orange, in Figure 3.13, show the nodes added to the open list as part of the recalculation, nodes B, N, M, V and K, show the new values after the expansion, again as the algorithm starts at the goal and works to the current position the updated values are relative to the goal increasing the cost the further away it becomes. The agent now just follows the path again until either the goal or an obstacle is found (Koenig & Likhachev, 2002).

D* lite eclipses the original D* with its simplicity and robustness reducing D*'s many nested statements into a clean one breakout condition and merging with A* efficiency (Koenig & Likhachev, D*Lite, 2002; Wooden, 2006).

Planning

Pathfinding is only one aspect of AI, while its important for the agents to successfully navigate the environment they are in there are other factors in game that might dictate behaviours. Agents may need to attack the player, interact with an object or investigate an area, without some kind of criteria to change the current goal, agents will be forced into completing the current task before searching for the next objective with no fluidity between them.

4. Finite State Machine

Finite State Machine (FSM) are a simple way to give an AI agent the appearance of making decisions. The behaviour of the agent needs to be defined, the original way to achieve this was to clearly define the states the AI could belong to, for example Attack, Flee or Heal that will dictate the behaviour of the agent (Millington, 2019). Once the states the agent could be in are defined the states need to have clear transitions, this could be an arbitrary value like spotted the player or use a curve to calculate how confident the agent is to attack (Wooldridge, 2024). However, due to the simple nature FSM's can suffer from state thrashing where the agent operates on the borders of transition and flicked between two states. For example, if the requirement to attack is the player is close enough to the player and the player is running away then the agent might try to attack, lose too much ground and transition to the chase state, catching up to the player at which point the agent will transition to attack and the cycle begins again.

5. Behaviour Trees

FSM's are extremely ridged and dictating that an agent can only be in one state at a time can cause state thrashing, where the agent is on the bounds of two states and will flicker between the two and not make a decision, as expected this can be jarring to the player breaking the immersion and fun of the game (Isla, 2005). Behaviour trees were developed as an improvement on the FSM creating a tree of behaviours that can be navigated based upon the agent's needs. Each time the agent acts the

this involves navigating through composite nodes, controlling the type of execution, Instruction nodes, control the flow back up the tree, and leaf nodes, the behaviour or task nodes (Roberts, 2022).

Figure 4.1 shows a basic Behaviour Tree layout with the different types of nodes. There are variations of composite and instructional nodes, each one designed to control the flow of decisions in different ways. Leaf nodes are the end of the branch, they cannot have any child nodes and contain the behaviour that is to be executed by the agent (Champandard & Dunstan, 2013). Instructional, also called Decorator, nodes can only have one child and can override the returned value from the leaf node, this could be inverting the result changing the success to failure or failure to success, or repeatedly calling the child node until a criterion is met. Composite nodes can have multiple child nodes and controls the execution of them, for example the selector composite node will execute all child nodes until one returns a success or all nodes have completed, whereas a sequence node will execute until the child nodes return a failure (Roberts, 2022; Millington, 2019).

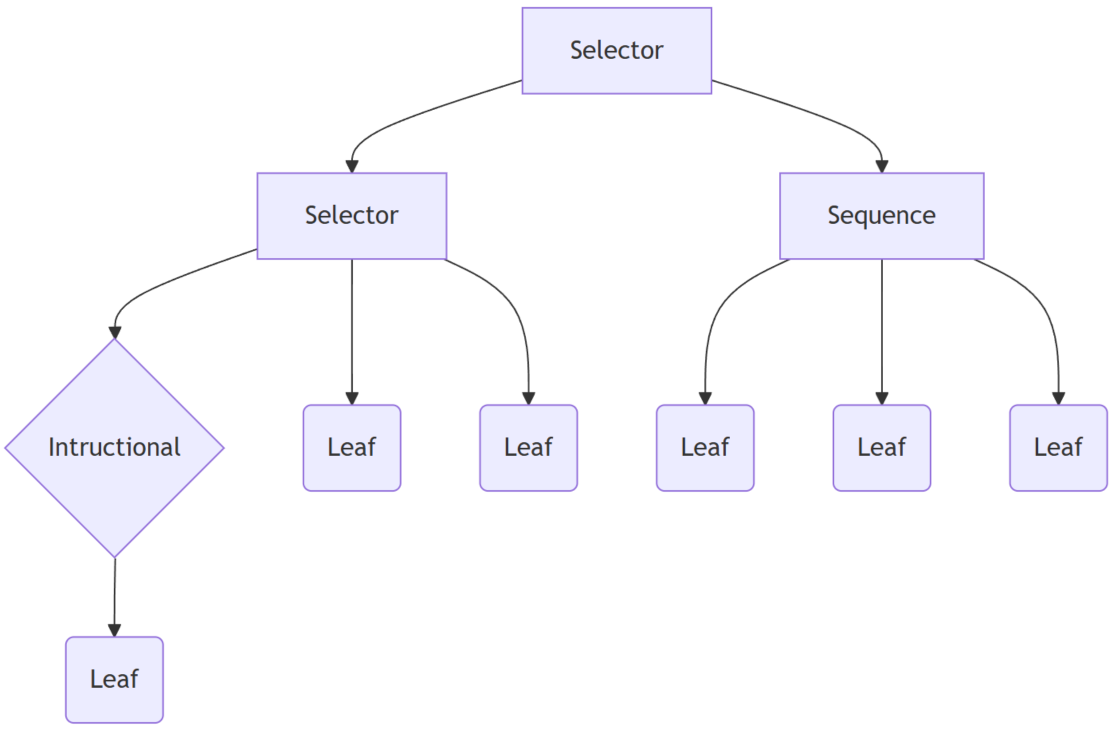


Figure 5.1 Basic Behaviour Tree Layout

Many Behaviour Trees share data between other trees using a Blackboard, this mean that data set in one tree can be used in another without having to pass the data between them. Blackboards achieve this be using a one to many relationship to allow many trees to be associated with the blackboard but only one blackboard for each tree this would allow AI agents that share a type could utilize the same data without duplicating many trees (Rallabandi & Roberts, 2024).

6. Reactive Behaviour Trees and MARPO

The goal of Behaviour trees is to improve upon the limited state of FSMs and give agents more adaptability however, in games the AI might need to react to new information, after taking a significant amount of damage attacking the player is no longer the priority but until the action is complete and the area reassessed then the agent might not heal. This all stems from the fact behaviour tress are natively not reactive, if an event is to be handled, the conditions leading up to it must be in the tree, leading to bigger and bigger trees that become difficult to debug (Rallabandi & Roberts, 2024). Reactive behaviour trees solve this by utilising some of the aspects of MARPO, Movement, Avoidance, Routing, Planning and Orders specifically the multiple stacks or queues to handle tasks, one to handle long-term planning events and one to handle any reactive or priority events, while not limited to just two data structures reactive Behaviour trees tend to use two (Rallabandi & Roberts, 2024).

MARPO takes the idea that task inherently have different priorities depending on the scenario and using different stacks plans for them. The example in figure 5.1 involves three stacks the first being the idle stack, the lowest priority, will contain the default behaviour of the agent, the action stack, the middle priority, contains the current tasks being carried out informed by the default behaviour in the idle stack. The final stack is the reactive stack and at the highest priority any tasks in this stack will be completed before anything else and only when empty will the agent return to its action stack (Bull, 2024a).

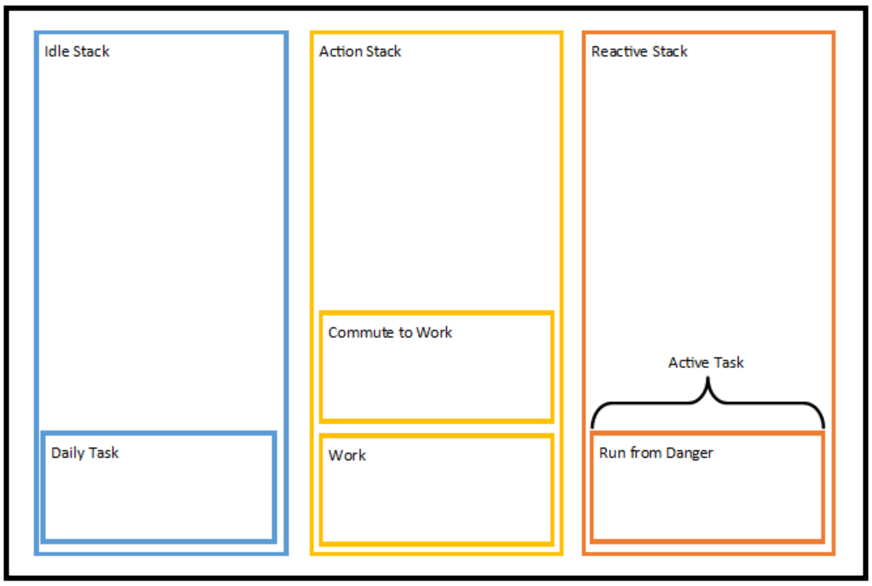


Figure 6.1 Basic MARPO task layout

As can be seen in figure 5.1 there is currently a task in the reactive stack that has become the active task disregarding what the agent was doing previously, these reactive tasks can come from anywhere in the program and is not limited to dangerous situations, it could be that the AI heard a noise and needs to investigate before returning to their default state. It's also important to note that if the reactive task is empty and all action tasks complete the idle stack will execute it's task which adds the work task to the action stack the work task then adds the commute to work task, essentially adding the goals required to complete that task to the stack (Bull, 2024a).

This reactive method has also been adapted for the use in simulations and Real Time Strategy (RTS) games by treating the stacks as a chain of command can inform the behaviour that is added to the stack. Figure 5.2 outlines how the stacks can be labelled for an RTS making the players actions the most important cements the feeling of command over the army, with the idle and action stacks breathing life into the agents (Bull, 2024a).

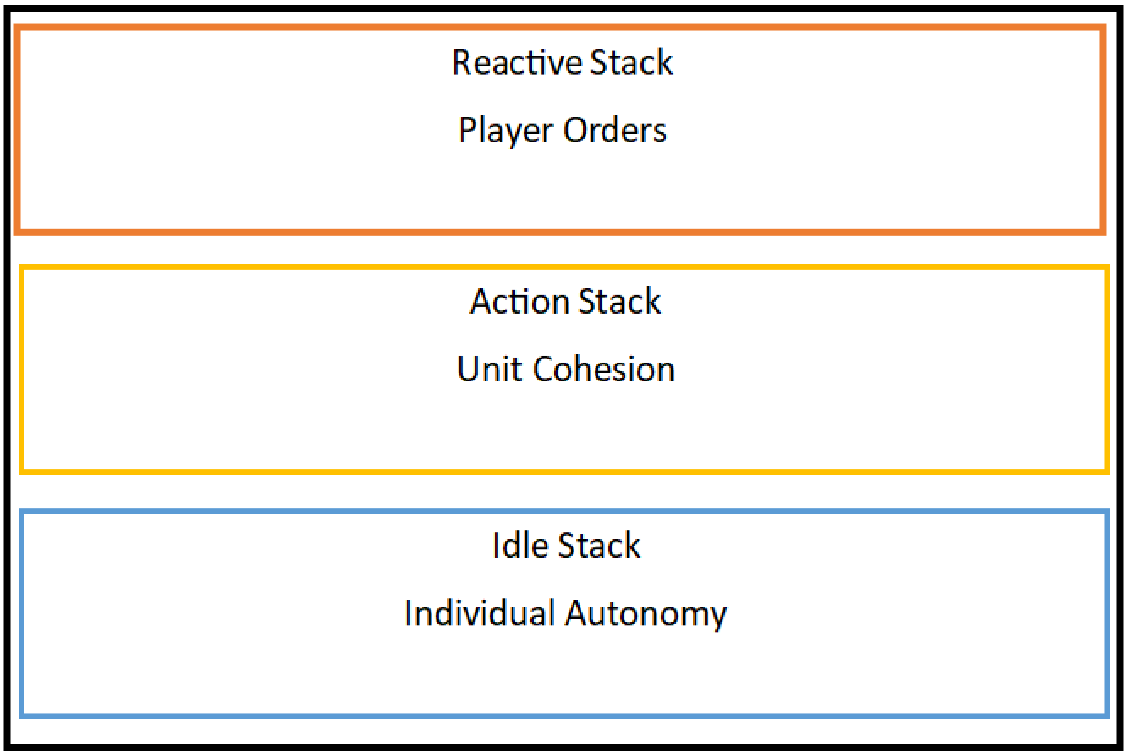


Figure 6.2 Order of priority in RTS games

7. Automated Planners

There have also been attempts to automate the process of planning actions to allow the AI agent to use the best strategy based upon the decisions that can be made, the most famous automated planning is Goal Oriented Action Planning (GOAP) (Condò, 2024). GOAP works on the premise of finding the best sequence of actions that will satisfy the agents long term goal, while a very compelling choice when it comes to AI planning GOAP has two major flaws the first being the number of possible combinations of actions, a naïve GOAP has the notation of $O(nm^k)$, the solution will take too long to provide an answer. The second is caused by creating a sequence of actions to take without considering the changing environment, this forces the agent to follow the actions without checking that the next action is still available (Millington, 2019; Condò, 2024).

Each of these decision-making architectures are designed to make agents reacted more dynamically; to appear intelligent to the player however, the level of reaction is determined by the overall vision of the game. These methods are also not isolated and can be combined for the desired behaviour, a simple FSM can become more reactive if merged with MARPO and the current state can be interrupted by a higher priority state.

Research Methodologies

To test how well an agent can navigate a changeable environment different pathfinding algorithms will be compared in different scenarios. The success rate of the pathfinding algorithms and planning method will be determined by how they satisfy these criteria:

- How efficiently a path can be found in several scenarios both static and dynamic. To draw conclusions from the testing different scenarios will be explored removing the bias of testing algorithms in areas that are not favourable against ones designed to solve that problem.

- How believable the paths taken are to participants. To test for believability participants will be asked to mark the path they would take to get from a start goal to the end goal, these entries will be correlated into a heat map, the path taken by the agent will be overlayed and given a score based upon how closely it matches.

- How adaptable the AI is when faced with an obstacle. It is important to this study of how the AI will react when met with an obstacle, be it player controlled (Locked Door), dynamic (Moving wall) or unknown when the path is calculated.

8. Artifact Creation

The artefact will be created using the SDL2[2], due to the level of control offered by the C++ library without focusing on the graphics pipeline. The game space will be split into a grid where each node can be given a cost to navigate, this approach will allow all pathfinding algorithms selected to function in the space with little to no modification, it also means that any map chosen or created can be easily divided into a uniform and repeatable pattern. Ensuring the overlaying of the map is repeatable is important for ensuring the tests are unbiased as with waypoints the placement depends on the goals and desires of the game, this might favour more open areas or obstacles and can affect a repeated test. Maps will be selected from Hotline Miami (Dennaton Games, 2013) to give a real example and generated mazes to test the algorithms computational power.

Testing which pathfinding method is best will involve incorporating several algorithms some which are designed for a static environment and ones created to be more dynamic to encompass a full range, the algorithms selected are Dijkstra's algorithm, A* and D* lite. Starting with Dijkstra's algorithm will be a good baseline for testing the effectiveness of the following methods, A* has been a staple in pathfinding for many years and should set the bar for the others, D* lite aimed to improve upon the workings of LPA* using an incremental search to deal with currently unknown obstacles. Exploring a range of different algorithms will also be important for testing the believability of the path found, while it is important for the path to be found quickly and efficiently if the players immersion is broken it has failed in believability.

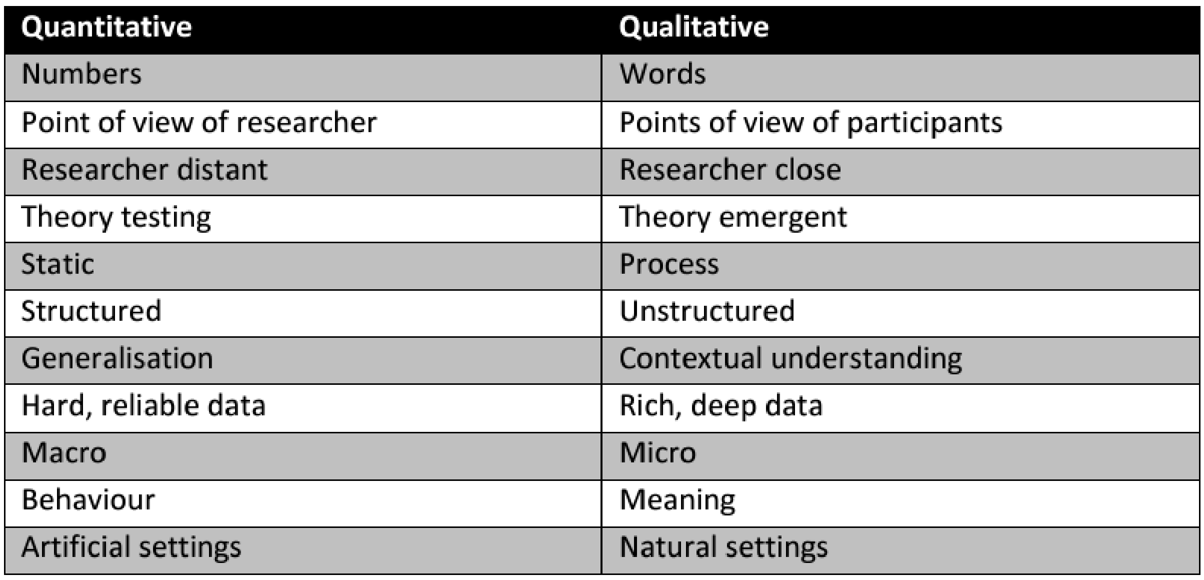
9. Testing and Results

To fully test the outcome of the pathfinding methods both computational data and optional data will be collected. Computationally this will be done by measuring the time taken for the path to be computed, completed and if needed recalculated.

The computational outcome will be processed into a data table showing a series of quantitative data that can be directly compared with the other methods and definitively answer which algorithm is best in the categories tested. However just using quantitative data only reveals half the picture as seen in figure 8.1 the table shows that quantitative shows the hard values through the eyes of the researcher, while qualitive shows the experience of the participant and when testing the path taken if a player does not find the

[2] SDL2 – Simple Directmedia Layer 2 [https://www.libsdl.org/]

path believable or it breaks immersion the speed in which the path has been found is secondary (Graue, 2016).



Quantitative	Qualitative
Numbers	Words
Point of view of researcher	Points of view of participants
Researcher distant	Researcher close
Theory testing	Theory emergent
Static	Process
Structured	Unstructured
Generalisation	Contextual understanding
Hard, reliable data	Rich, deep data
Macro	Micro
Behaviour	Meaning
Artificial settings	Natural settings

Figure 9.1 Table of Quantitative vs Qualitative (Bell, et al., 2015)

To test where the path is believable the opinions of the player(s) must be considered, to this end a survey will be created giving participants the option to express whether they would take a given path or if they would find it believable. It is important to note that while the player may take the path, they might not expect an AI agent to do the same and conversely the path might be believable, but the participant would never go that way. The survey will contain two parts the first will involve participants drawing on the maps the path they would take from a given start to the goal. The results will be correlated into a heat map and the AI paths overlayed to see how close the path taken matches with the participants, by taking the nodes the participants travel through and calculating the percentage of those the agent uses. The participants for this study will remain anonymous but will have a background in game development, this maybe students, lecturers or people in industry.

The second will be a basic questionnaire that allows the participant to express their thoughts on how believable a given path is, for each of the pathfinding methods. This will allow the participants to rate the paths the agent has taken on scale of how believable they find it; they will then rate the paths against each other on which they would be most likely to take. It is important that the participant completes the heatmap first as to not be influenced by the paths seen in the questionnaire.

10. Interpreting the Data

Computational Data

To ensure a clear comparison between the pathfinding algorithms can be established, a graph will be generated for each map showing the time taken to find the path on each map. Each pathfinding algorithm will be invoked three time to establish a pattern and an average taken to show which algorithm was fastest. With the goal of games to reach an average frame rate of 60 a second it would be important to know which of the algorithms implemented are the most efficient as taking too long to find the path may take the method out of contention.

Questionnaire

To appropriately gauge the believability of the algorithms the participants results will be used to create a series of graphs denoting the average believability score between one and fiver. The higher the score will give the algorithm in question a higher consideration when combined with the efficiency testing. An additional graph will be created based upon the participants ordering of paths they would take, comparing this with the paths they found believable the results may reveal differences between what is acceptable for an agent to take vs what is expected when a player travels.

Heat Map

To compare the paths drawn by the participants to those taken by the AI a metric must be established, overlaying all the paths taken for each map and marking against the path found by the agent to determine how similar they are.

Results and Findings

Investigating the claim that dynamic pathfinding solutions can be used in games without breaking the players immersion led to the creation of several charts depicting data gathered.

11. Computational Results

Each map will adhere to the follow rules:

- All solid green squares are non-negotiable.

- Goal is denoted by the red square
- The blue square is the agent
- Orange squares show the path found

Figure 11.1 shows the first maze used for testing the time taken by each algorithm, due to the simple nature of the maze only having one correct path though made it perfect for comparing time taken.

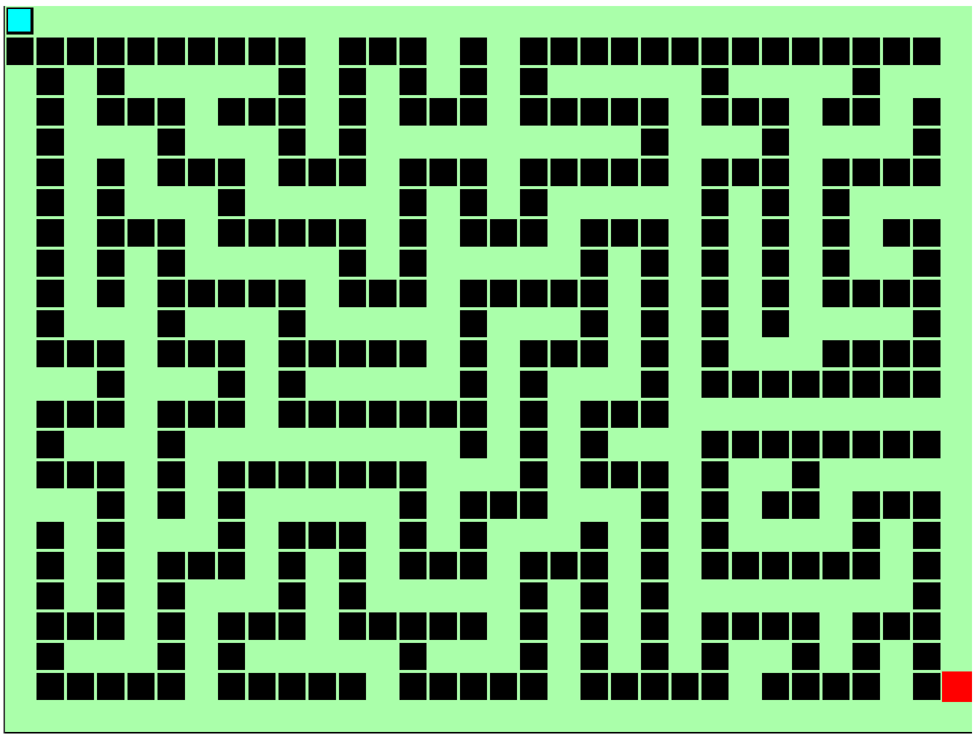


Figure 11.1 Maze One Map[3]

The chart in Figure 11.2 show the timings each pathfinding algorithm took to navigate the maze. Interestingly Dijkstra's algorithm outperformed both heuristically driven pathfinding techniques with and average time of 0.023ms and A* performing the worst out of the three with an average of 0.046ms. Naturally as there is only one path to the goal all three pathfinding algorithms found the correct path to the goal in an appropriate amount of time.

[3] AI Maze Generator [AI Maze generator]

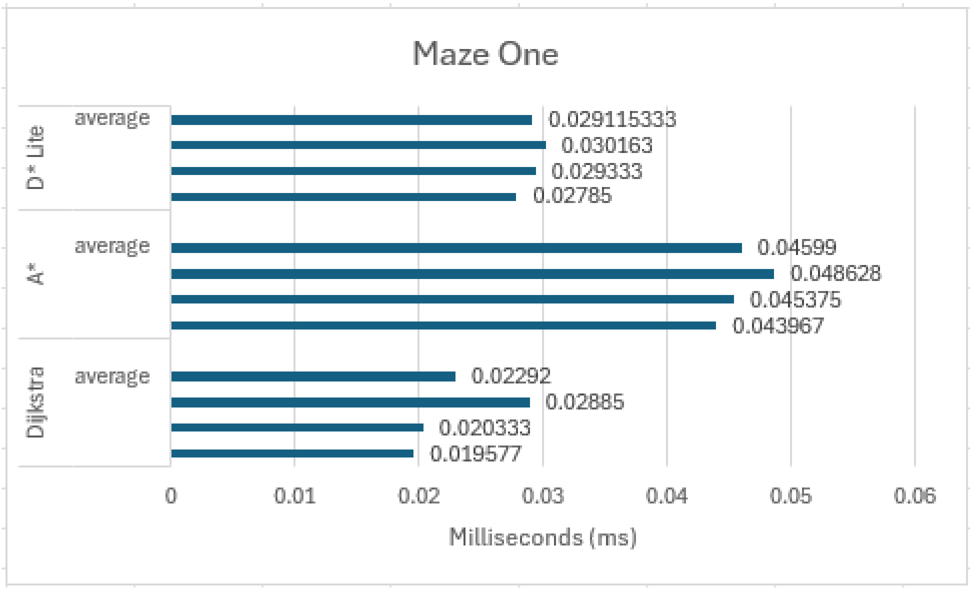


Figure 11.2 Maze One Results of Testing

Figure 11.3 contains the map of maze two used for testing, similar with maze one the there is only one route through the map. However, the start and end has been adjusted to the middle of each wall on opposite sides.

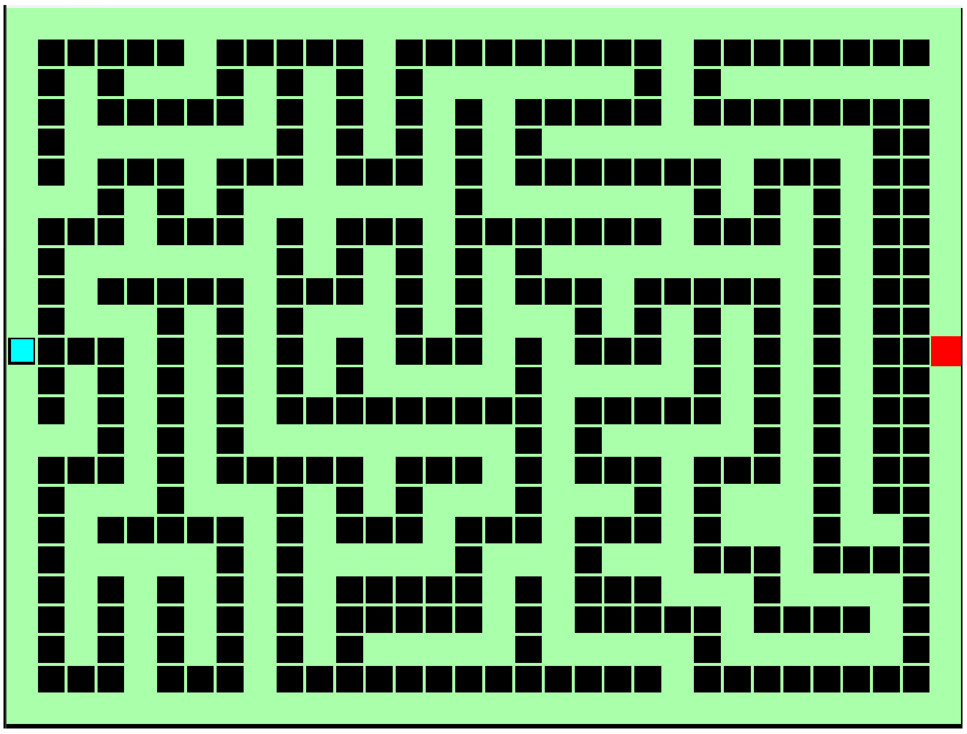


Figure 11.3 Map of Maze Two[4]

[4] AI Maze Generator [AI Maze generator]

The results of testing maze two are held in Figure 11.4 showing Dijkstra's algorithm is the fastest with an average time of 0.0064ms. However, unlike maze one D* lite proves the slowest with an average of 0.041ms.

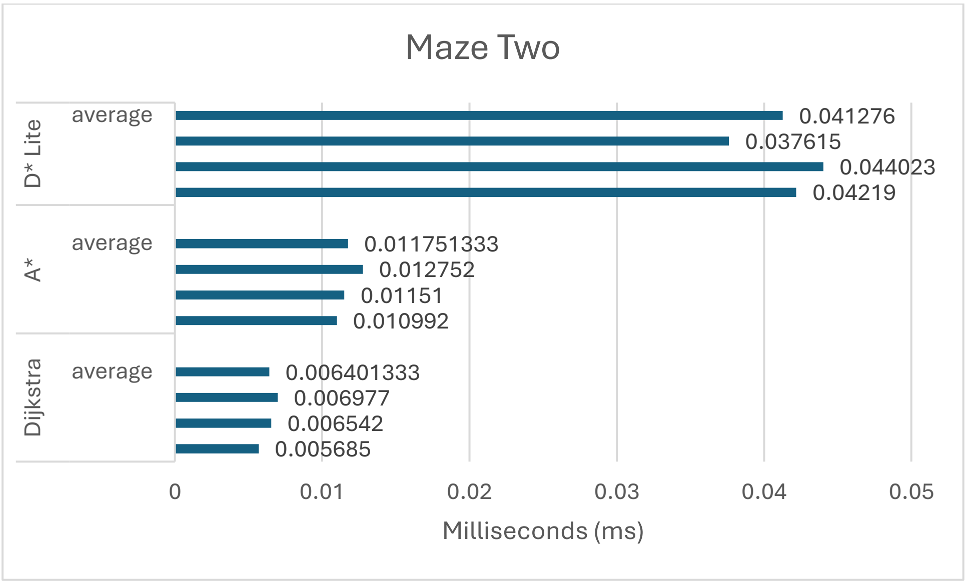


Figure 11.4 Maze Two Results of Testing

The first game map tested was Decadence from Hotline Miami (Dennaton Games, 2013) shown in Figure 11.5, the start and end point of each game map remained the same between the computational and participant testing.

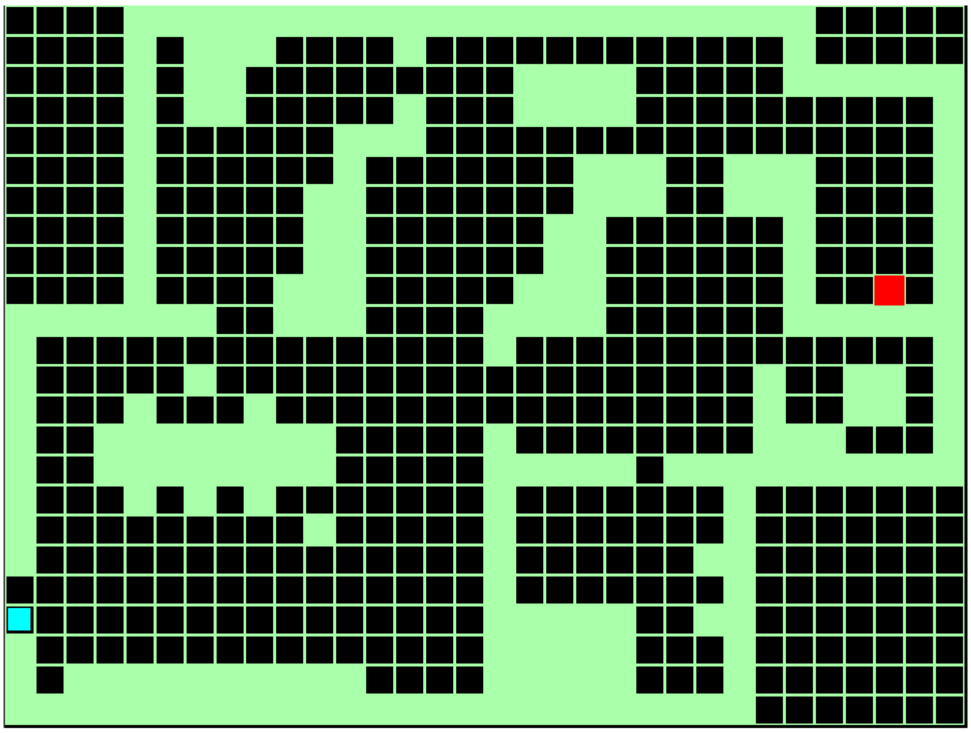


Figure 11.5 Map of Decadence from Hotline Miami (Dennaton Games, 2013)

Figure 11.6 shows the results from testing Decadence, in which the timing between the algorithms is very similar with Dijkstra still proving the fastest on average with 0.07ms and D* Lite being the slowest with an average of 0.086ms.

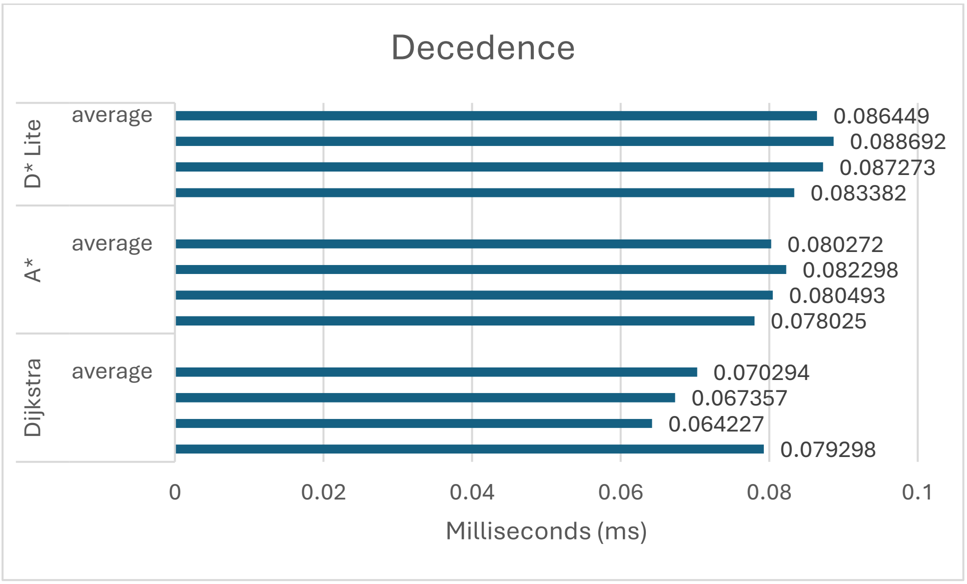


Figure 11.6 Results of Testing Decadence

The map bellow denotes the No Talk level from hotline Miami (Dennaton Games, 2013).

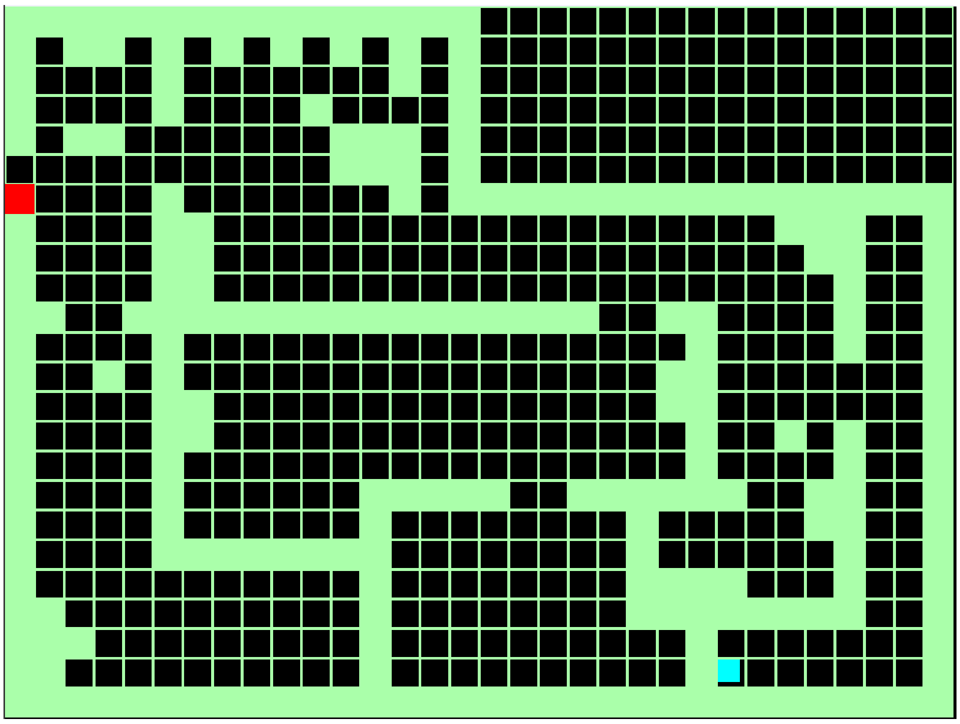


Figure 11.7 Map of No Talk from Hotline Miami (Dennaton Games, 2013)

Test No Talk shown in Figure 11.8 show A* to be significantly faster than the other algorithms with the average time of 0.037ms and D* Lite the slowest with an average of 0.087ms to find the path.

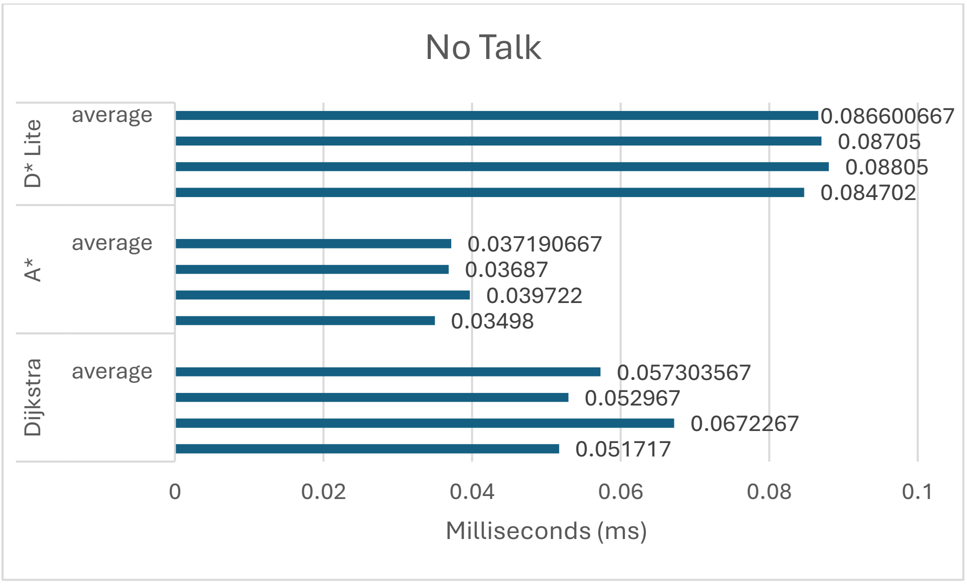


Figure 11.8 Results of Testing No Talk

The Metro map is shown in Figure 11.9.

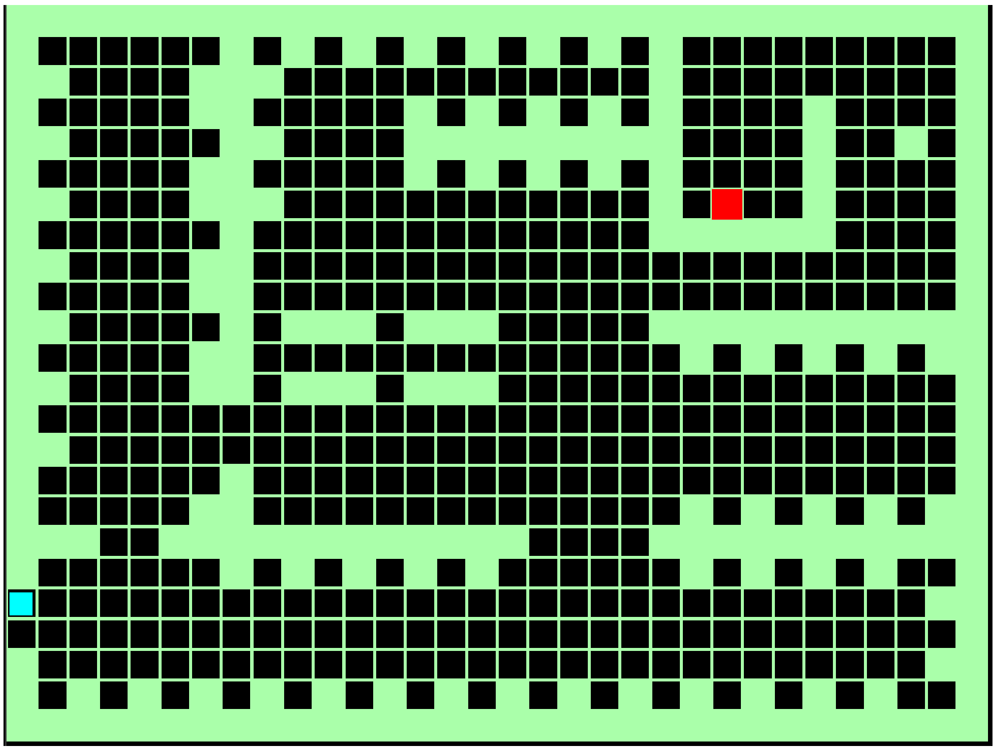


Figure 11.9 Map of Metro from Hotline Miami (Dennaton Games, 2013)

The chart in Figure 11.10 shows that for Metro D* Lite was the fastest with an average of 0.081ms and A* was the slowest with an average of 0.13ms.

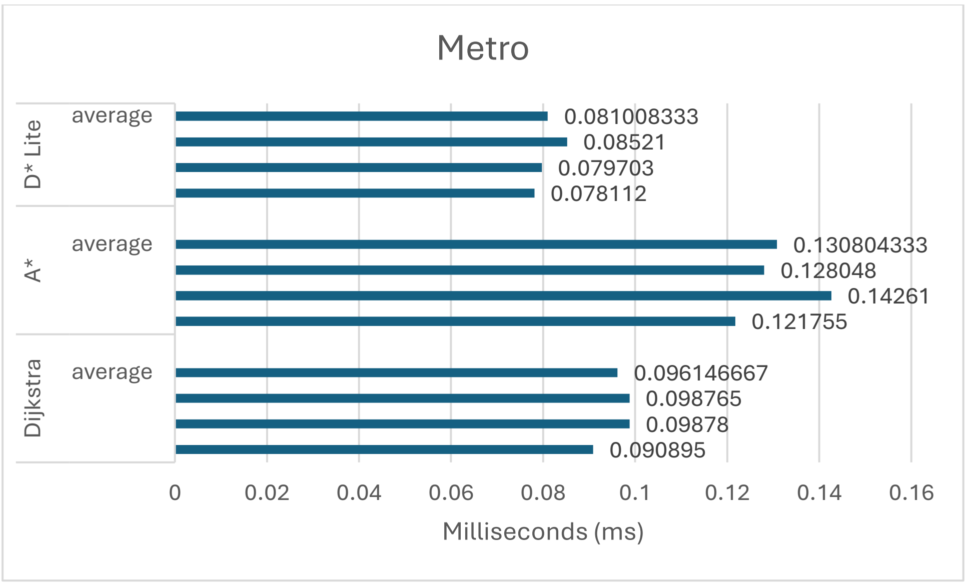


Figure 11.10 Results of Testing Metro

12. Heatmap Results

All participants routes are marking in white in the following maps with nodes outlined in orange representing nodes searched by the algorithm but not followed.

Heatmap: Decadence

The Heatmap in Figure 12.1 shows the paths taken on Decadence overlayed with Dijkstra's algorithm. Its clear to see in the opening stages many of the responses match that of Dijkstra however toward the middle most veer off leaving 29% of the paths following close with the agent's route.

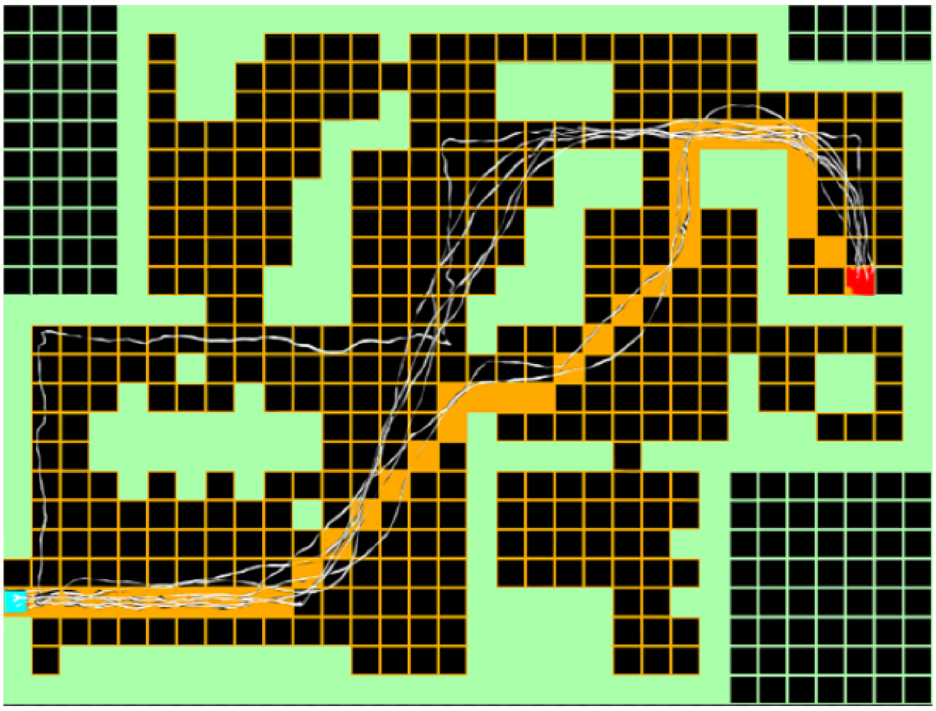


Figure 12.1 Decadence Heatmap with Dijkstra's Algorithm

Figure 12.2 shows Decadence with A* overlayed, presenting that A* has a much stronger finish then Dijkstra's following the heatmap very closely at the end achieving an almost exact match. However, the start A* diverts from the heatmaps completely not connecting with any similar nodes until rejoining them in the middle.

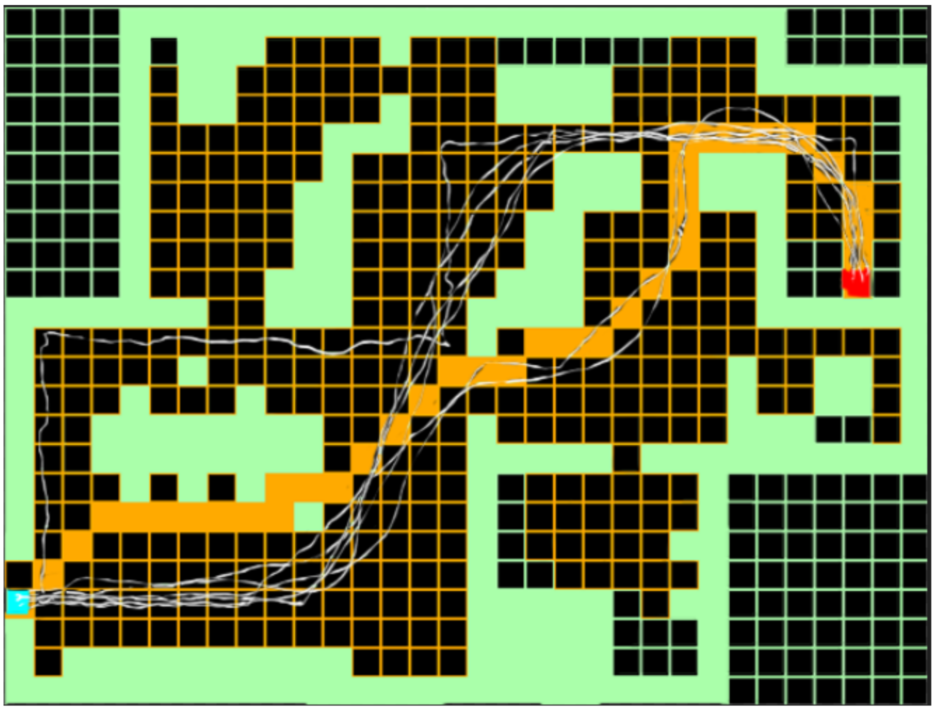


*Figure 12.2 Decadence Heatmap with A**

Decadence with D* Lite shown in Figure 12.3 outlines a very strong start following 85% of participants and continues to follow till the centre where it diverges and only follows 29% of participants before regrouping with the rest. D* Lite also diverges at the end choosing to cling to the wall.

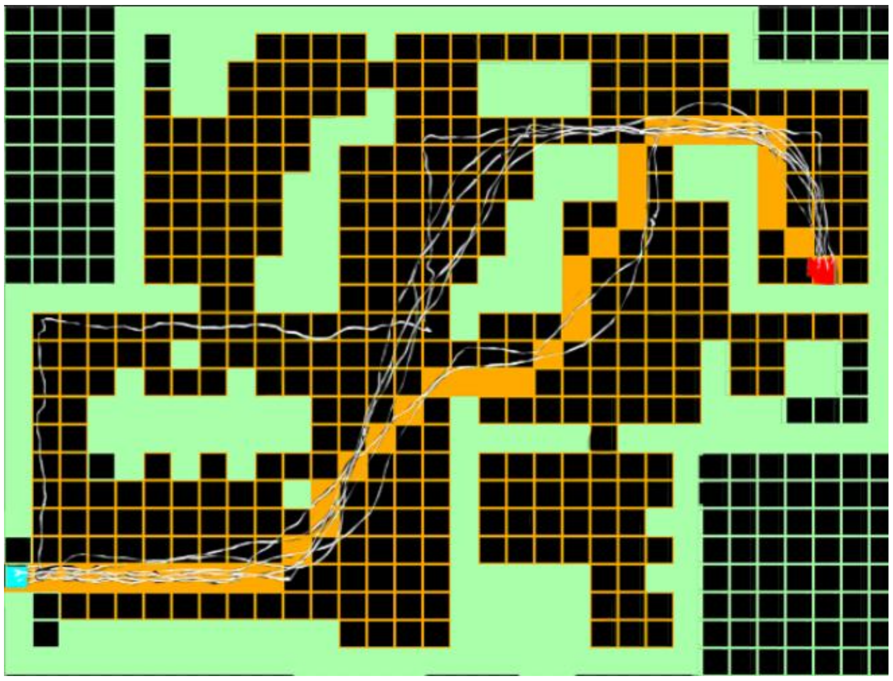


Figure 12.3 Decadence Heatmap With D Lite*

Heatmap: No Talk

Dijkstra's path through No Talk presented in Figure 12.4 shows a close match between most paths drawn by participants with only two paths diverging close to the middle of the path. This map also contains an outlier path that searches along the top of the map diverging from all other participants.

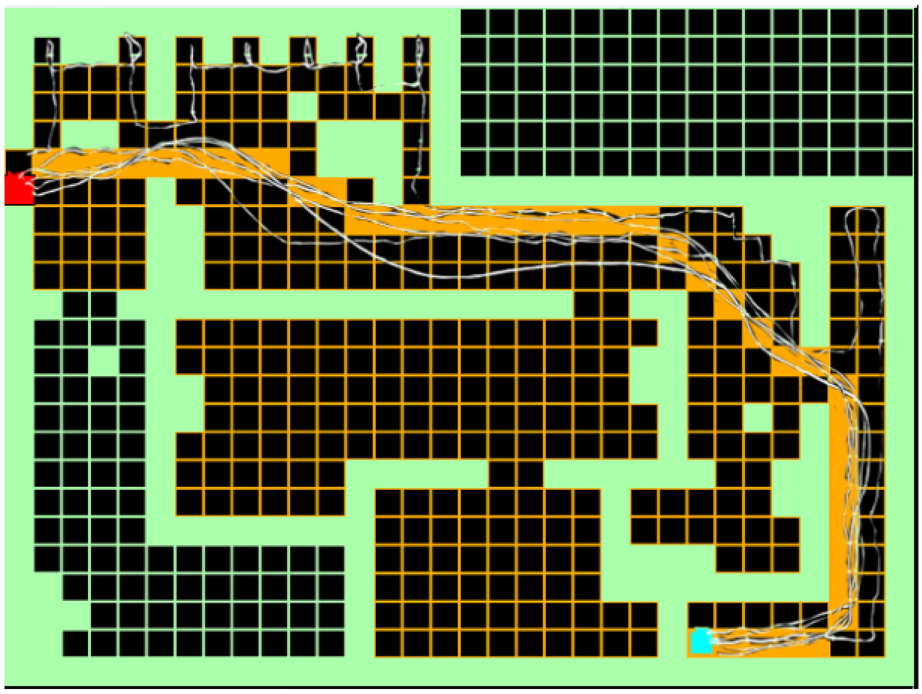


Figure 12.4 No Talk Heatmap with Dijkstra's Algorithm

Figure 12.5 shows A* pathing through No Talk, A* follows a more averaged path through the level only diverting from 29% of paths through the middle. Following closely until the end A* clings the wall closest to the corner while the participants tend to favour the other side.

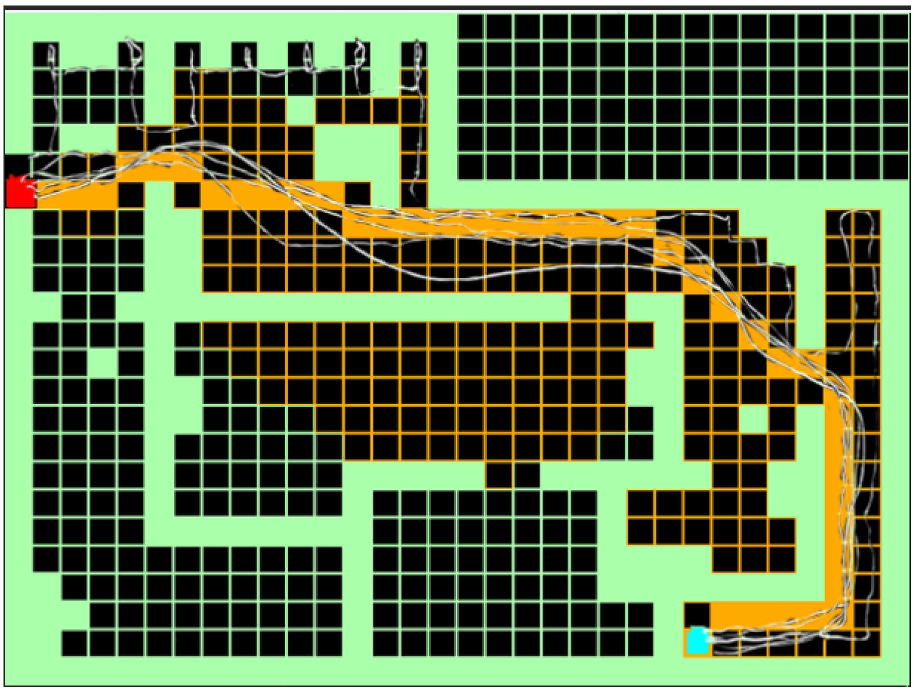


*Figure 12.5 No Talk Heatmap with A**

D* Lite deviates the most of No Talk shown in Figure 12.6 only matching 14% of the paths taken through the centre of the map. However, towards the goal D* Lite rejoins the group matching the majority of paths.

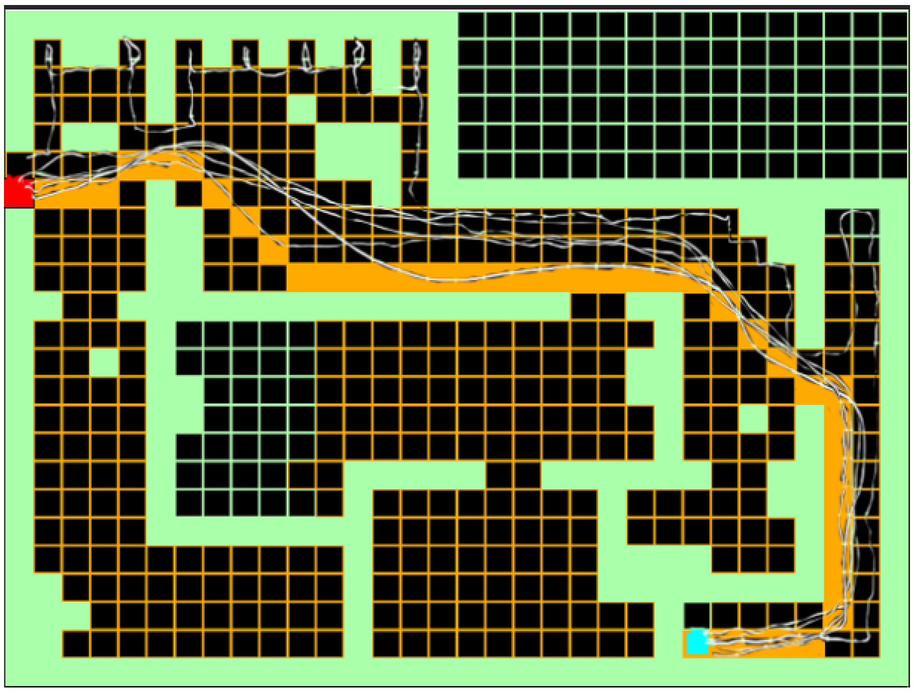


Figure 12.6 No Talk Heatmap with D Lite*

Heatmap: Motel

Figure 12.7 outlines the paths taken on Motel with Dijkstra's algorithm. The path found follows 57% of paths going left around the first obstacle while the other 43% choose to go right. Closer to the end is where the participants paths begin to diverge as some prefer to stay closer to the wall then others.

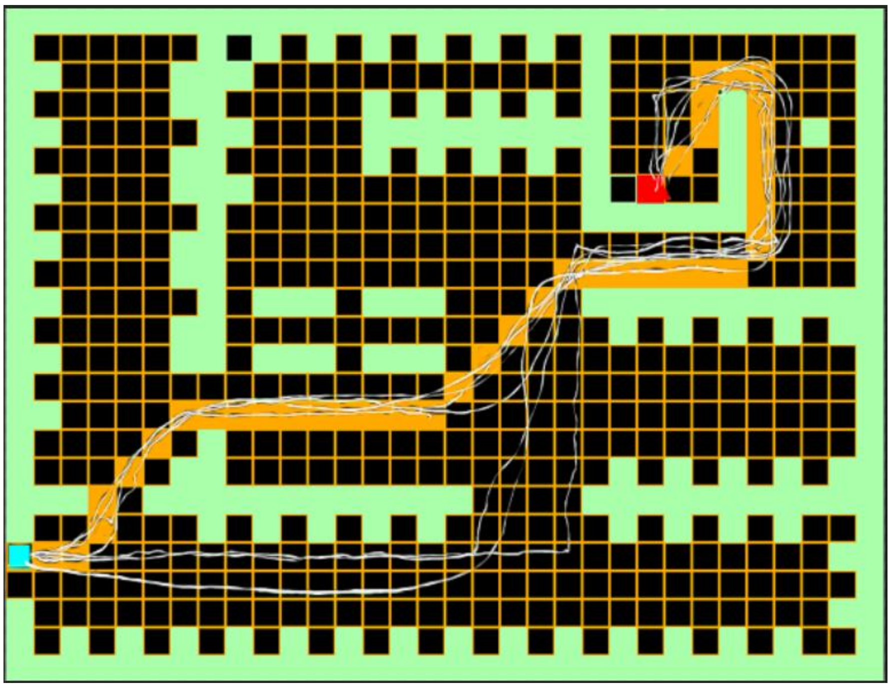


Figure 12.7 Motel Heatmap with Dijkstra's Algorithm

Figure 12.8 shows the path taken by A* and while it follows the majority it tends to favour sticking to the walls whereas the participants like to walk in the centre. Towards the end A* follows general path better than Dijkstra's choosing to stay away from the wall.

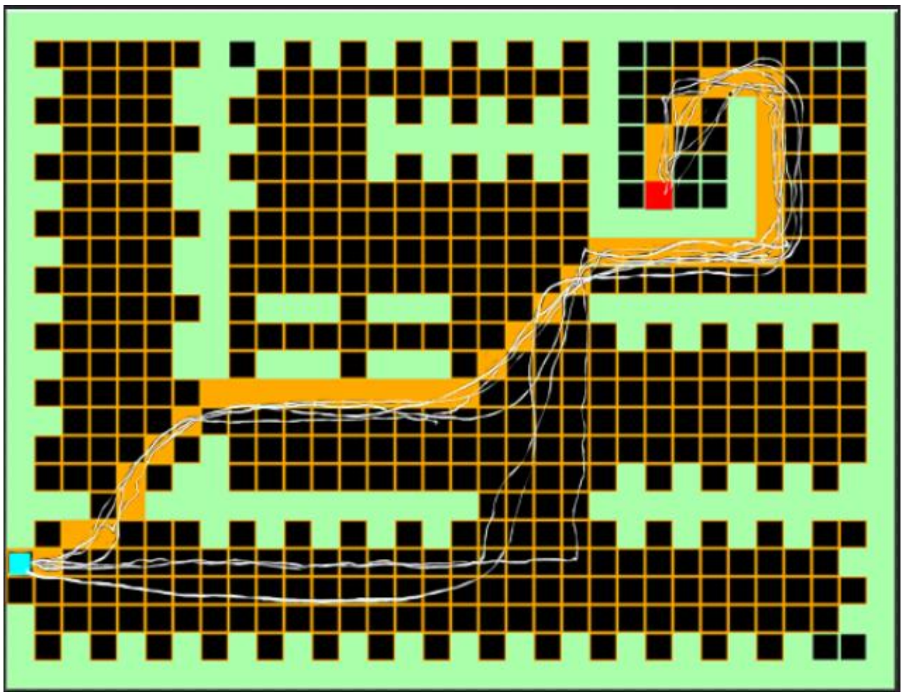


*Figure 12.8 Metro Heatmap with A**

D* Lite follows the same principles as the previous methods shown in Figure 12.9 D* Lite goes left around the first major fork but unlike A* stays away from the wall toward the middle of the route and sweeps around at the finale.

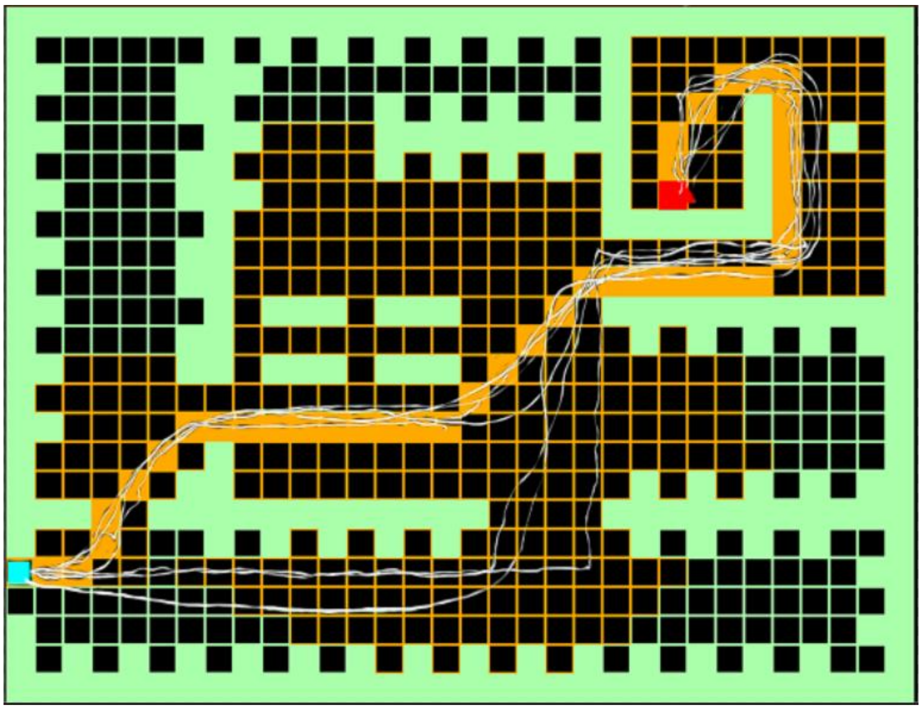


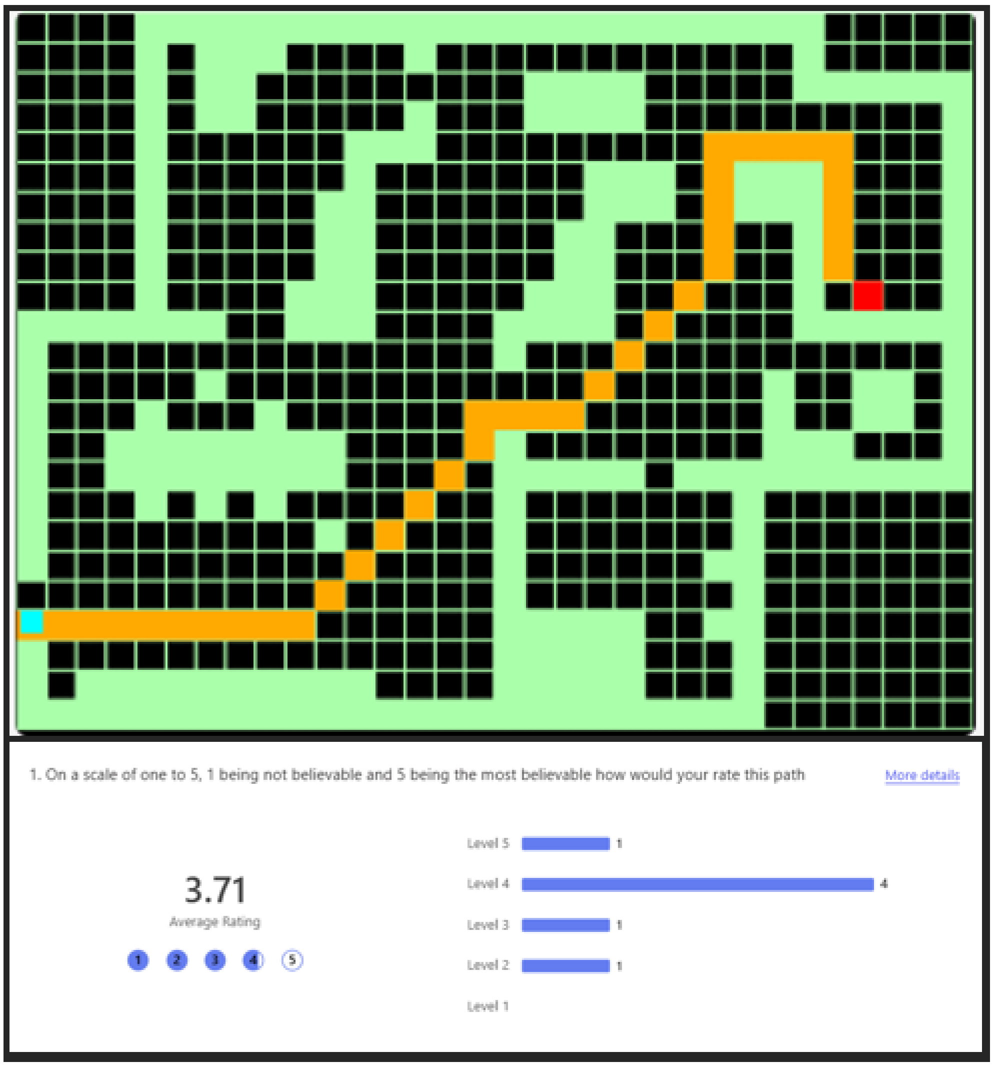
Figure 12.9 Metro Heatmap with D Lite*

13. Questionnaire Responses

The questionnaire outlined in Appendix 1 was presented to participants for their input. The results of the questionnaire are listed below.

Question One – Dijkstra's Algorithm

The feedback from question presented in Figure 13.1 Results of Question One of Questionnaire many participants gave the path found by Dijkstra's algorithm a score of 4 but with a low of 2 the final score is 3.71.



1. On a scale of one to 5, 1 being not believable and 5 being the most believable how would your rate this path

More details

3.71

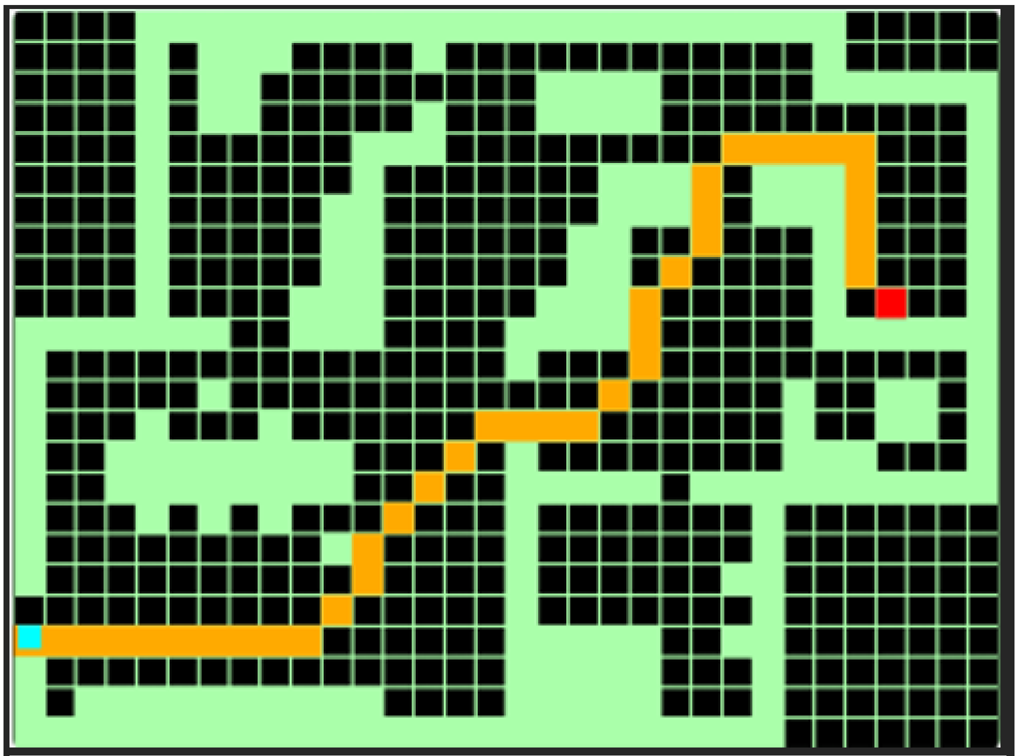
Average Rating

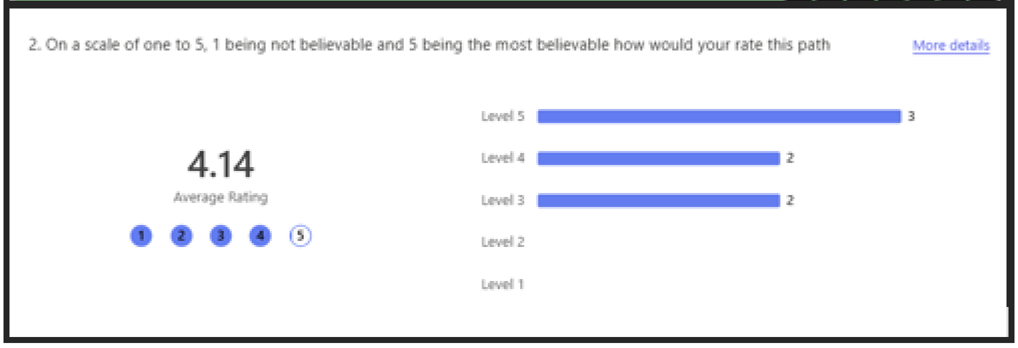
Level 5	1
Level 4	4
Level 3	1
Level 2	1
Level 1	

Figure 13.1 Results of Question One of Questionnaire

Question Two – D Lite*

The second question involved the path found by D* lite shown in Figure 13.2 Results of Question Two of Questionnaire, participants rated the believability an average of 4.14 out of 5 and with a tight range of values between 5 and 3.





2. On a scale of one to 5, 1 being not believable and 5 being the most believable how would your rate this path

More details

4.14

Average Rating

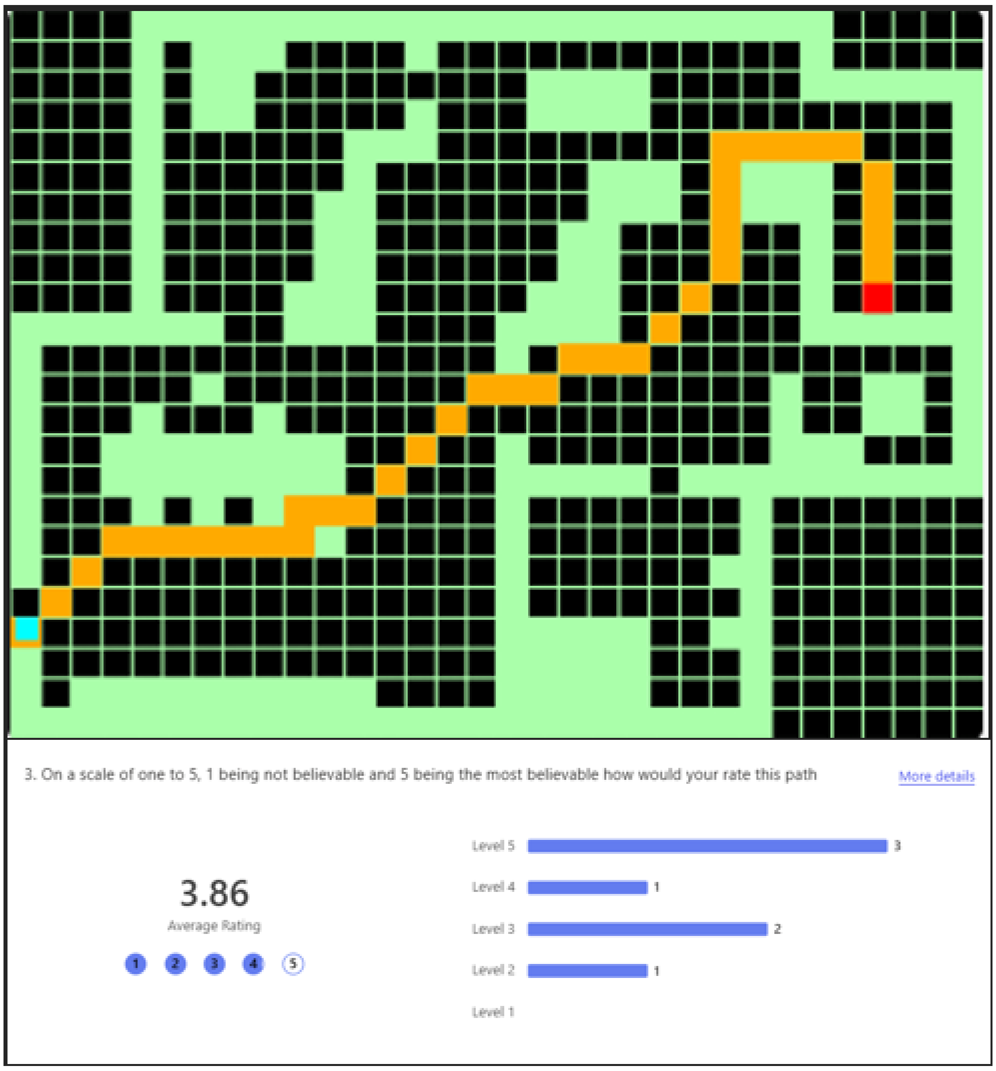
① ② ③ ④ ⑤

Level 5	3
Level 4	2
Level 3	2
Level 2	
Level 1	

Figure 13.2 Results of Question Two of Questionnaire

*Question Three – A**

The final path presented was that of A* shown in, the average believability score given by participants was 3.86 with scores ranging from 2 to 5 out of 5.


3. On a scale of one to 5, 1 being not believable and 5 being the most believable how would your rate this path

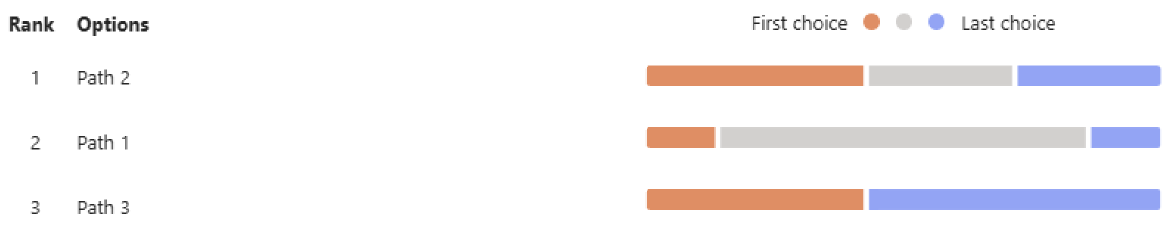
More details

3.86
Average Rating

Figure 13.3 Results of Question Three of Questionnaire

Question Four – Ranking of Paths

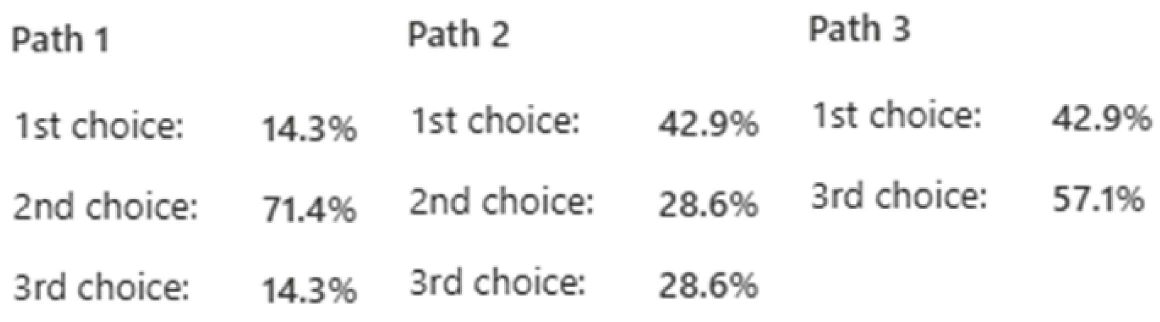
The final question of the questionnaire involved the participants putting the previous three paths in order of which they are most likely to take then the second and finally the path they are least likely to follow. The results presented in Figure 13.4 shows the overall order the paths where chosen in.



Rank	Options
1	Path 2
2	Path 1
3	Path 3

Figure 13.4 Results of Ranking the paths

Figure 13.5 outlines the full breakdown of the results revealing that paths 2 and 3 are chosen as the first choice the most with 42.9% of participants ranking them highest. Path 1 dominates the second choice with 71.4% of participants choosing this as the first backup path. Finally path three is rated the lowest being chosen last 57.1% of the time but also interestingly not chosen as any participants second choice.



Path 1		Path 2		Path 3	
1st choice:	14.3%	1st choice:	42.9%	1st choice:	42.9%
2nd choice:	71.4%	2nd choice:	28.6%	3rd choice:	57.1%
3rd choice:	14.3%	3rd choice:	28.6%		

Figure 13.5 Breakdown of Results of Ranking the paths

Final Thoughts

Correlating the results together shows D* Lite being many participants favoured choice of path taken, combined with its position being the most believable, rated 4.14/5, puts it in high standing. The results also outline that Dijkstra's algorithm performed better than expected, beating both the other algorithms on several maps for efficiency and providing a steadfast second choice for paths taken. While A* being the staple tended to fall behind its counterparts without making much impact.

Discussion and Analysis

14. Computational Discussion

Mazes

The inclusion of heuristic's aimed to increase the efficiency of the pathfinding however, in these cases involving the maze's both A* and D* Lite were slower at producing a path compared to Dijkstra's algorithm. This is likely due to how limited the search space was Dijkstra's limitation of searching in all directions was hindered greatly reducing the number of nodes that could be searched. The reduction of the search space also meant that Dijkstra's weakness of not knowing where the end was also meant that an extra calculation was not done for each node that was searched.

In both maze maps Dijkstra's algorithm proved to be the fastest however, in Maze One, D* Lite beat out A* but on Maze Two A* is faster than D* Lite. It is said that in most cases D* Lite is just as fast as A* (Koenig & Likhachev, 2002) it is likely the setup of the maps that has caused this. Figure 14.1 investigates this further the maze on the left shows A*'s approach to the maze and the nodes searched, outlined in orange, are heavier at the start as the paths diverge but head in the direction of the goal. The image on the right shows D* Lite and the number of nodes searched is greater towards the goal, this is because D* Lite works from the goal towards the agent rather than A* which works from the agent toward the goal. This means that when going from the goal toward the agent there are more diverging pathways, they algorithm needs to search. A potential fix to this problem could be searching from both the goal and the agent simultaneously and meeting in the middle, while this will present its own set of challenges it might be able to counteract the inefficiency encounter on certain map layouts. This proposed change may also aggravate the current issues with the algorithms getting locked into an incorrect route.

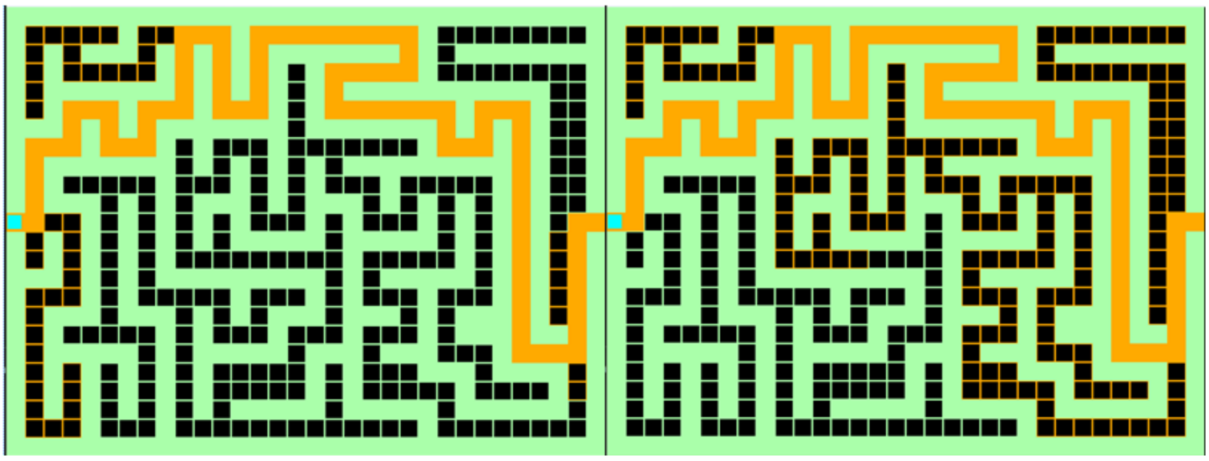


Figure 14.1 Maze Two Paths A(Left) and D* Lite(Right)*

Game Maps

When it comes to the Hotline Maimi (Dennaton Games, 2013) maps there is a range of different results. Each pathfinding algorithm beats the others for fastest on one map each. With this range of results, it tricky to definitively say which is the fastest, however it raises the consideration that pathfinding algorithms or heuristics likely favour a particular layout. As proven by the mazes the number of diverging path and how restrictive the map is tending to play to the strengths and weaknesses of the algorithms making some better and others inefficient. To confirm this a wider range of heuristics and maps could be used, as it is unlikely there will be one answer the problem but there may be a best-case solution that provides the best results in most scenarios.

15. Heatmap Discussion

Each of the heatmaps created can be interpreted different ways as the player is the most dynamic object in any game scene and without bounds the participants were able to draw any path they liked if they started at the blue square and ended on the red.

Decadence

The Decadence heatmap shows that most players stuck together moving to the centre of the map avoiding the big obstacles until they must navigate around them. However, in the middle of the route the participants diverge the route taken some choosing to continue the current bearing heading left and others divert around the right. While moving around towards the right is more efficient only 29% followed this route, thusly doesn't hold the majority.

A* also deviates from the path immediately deciding that it is more efficient to head diagonally towards the first obstacle where all participants stayed clear leaving A* with the lowest score out of the gate but finishing stronger than the other two algorithms.

No Talk

No Talk is a much tighter map with less space to navigate this did not stop the participants attempting to steer clear of obstacles as much as possible with some taking wide berths around corners and 43% of paths aiming to stay in the middle of corridors and avoid clinging to the walls. Due to the restrictive nature of the map both Dijkstra and A* overlapped with many participants whereas D* Lite clung to the opposite wall. The wall that the algorithms follow tends to depend on the direction that the path travels in all maps A* and D* Lite cling to opposite walls. It would be interesting to get participants to navigate from the goal to the player instead to see if the direction of travel also changes the player's path.

Due to the deviation in the D* Lite path it only matched 14% of paths taken and for some sections didn't match any making it the least accurate on this map.

Metro

Metro showed a change in the players paths, in the previous two maps many participants moved forward without deviating until the first obstacle. 57% of players rerouted immediately around the obstacle, something that was copied by all pathfinding algorithms. D* Lite and A* disagree again on sections of the path however, unlike previous results the players switched between the walls they clung to. Early on D* Lite successfully matches with 57% of participants but later drops down to 14% whereas A* does the opposite matching with 29% early one but merging with the rest of the participants later with 86% matching.

Heatmap Conclusion

The movements of the player through an environment maybe more psychologically charged then initially thought, while D* Lite and A* disagreed on which wall to cling to the participants developed a pattern of avoiding any obstacle as much as possible. Changing the cost of traversal for nodes that have a wall adjacent might provide a path that closer resembles the paths taken by the participants.

16. Questionnaire Discussion

Question One – Dijkstra's Algorithm

Many participants rated Dijkstra's algorithm 4 out of 5 claiming 57% of the vote giving a very high confidence level. However, because of the range of scores given to Dijkstra the average rating comes to 3.71/5 giving 74.2% confidence rating that participants would believe the path created by the algorithm. However, with 14% giving a score of 2/5 it outlines that the minority that are not confident with the believability are almost certain that they wouldn't believe this path.

Question Two – D Lite*

D* Lite was rated the best out of the three pathfinding algorithms, scoring an average rating of 4.14/5 making its confidence value of 82.8%. The lowest score given was 3/5 making the score achieved more concrete with most participants that over 50% of the time they would believe the path taken.

*Question Three – A**

A* had a full range of scores starting at 2/5 all the way up to 5/5 with the majority putting 5/5. Although most people put a top score the second most popular score was 3/5 bring the average score down to 3.86/5, 77.2% confident, making it less believable than D* Lite but more believable than Dijkstra's algorithm. Despite the high confidence value, it is hard to confirm that this path will not be questioned due to the full range of scores given there is a probability the path will make some player question how believable the agent is.

Question Four – Ranking of Paths

In the test of believability Dijkstra's algorithm scored the worst with the lowest confidence level however, when asked to rank the paths in order of which the participant is most likely to take to the lowest Dijkstra was many participants second choice. Dijkstra was selected 71.4% as the second path selected and on 14.3% for both first and last choice, meaning that despite the low confidence many believed it better for the player to take then one of the others.

D* Lite and A* was put in the top spot by 42% of the participants, tying them perfectly as the paths most likely to be taken. D* Lite was also rated at the second and last choice slots for 28.6% each, despite it taking the top spot so often participants ranked Dijkstra as more favourable for the second choice.

While A* contested D* Lite as most likely to be taken by a player and beat Dijkstra in the believability test 57.1% of the time A* was put in the least likely slot and no score for second choice. This could mean that players would not question if an agent took that path but acting as a player in the game are unlikely to follow the path that is laid out by A*, and if the agent is expected to move like a player this could break immersion (Roberts, 2024b).

Conclusion

The believability of any given path is not as clear as it might seem, it is easy to see a problematic behaviour when an agent walks into a wall, but harder to understand what behaviours the player might question. Each player will develop an understanding of how the game world is expected to work and will adjust their tolerances accordingly. This understanding can be seen in the heatmaps where some participants took the nature of the grid to mean they could only move between the squares directly, or others looking to satisfy an exploration path.

There is a distinction between how a player expects another player to move then that of an AI agent. In the real-world people will take alternate routes based on knowledge of the environment. This could be physical like avoiding roadworks but could also be an emotional avoidance, taking a route that avoids a road they do not like. This route planning transfers into the game world seen in testing participants would take differing routes that may be slower but based on their preference more appealing.

It is more important to understand the players expectation of what is believable then to try to account for every possibility. Every pathfinding algorithm presented builds upon the previous; Dijkstra's algorithm formed the basis that formed A* and adaptation of A* created D* Lite. However, the progression is not always a straight line, it is precisely the advantage of A* that limited it in some of the tests, searching for a route with heuristics guided A* and D* Lite down the wrong paths thinking they were getting closer.

The efficiency of pathfinding algorithms are just as constrained by their environment as much as they are the algorithm itself. The current layout of the grid and the maps selected favour the simpler algorithms, by limiting the number of nodes that can be searched in an area the weakness of Dijkstra's algorithm cannot be fully realised and inadvertently the benefits of heuristics are prevented. This changes the selection process of pathfinding algorithms to match the environment to be searched, as space complexity, as much as it is to make the algorithm more efficient in time complexity.

Reactivity can give an agent an advantage over more static methods of creating AI behaviours however, believability must be considered in all aspects of the development process. Making the most reactive agent with human-like behaviours without factoring in believability on how they could navigate the space could impact the level of Flow the player can experience. The impact of an agent running in fear is dampened if that agent gets stuck walking into an obstacle.

This would mean that there is a potential for games to use an ineffective algorithm for the search space, while testing pathfinding algorithms in a small search space it was found that each algorithm found strengths in different types of maps. Condense maps such as mazes allowed Dijkstra's algorithm to find paths quickly and effectively the search space itself controlling the number of nodes that can be searched. However, in spaces that had more open areas the use of heuristics forced the search in the direction of the goal. This was not always the case when the direction the search travelled in as it allowed the path to get stuck searching down paths that ended abruptly.

There is also a trade of for the believability of the path that is generated. Discovered during testing the most believable pathfinding algorithm D* Lite was generally the slowest algorithm on the maps selected. While the least believable Dijkstra's Algorithm, proved to be the fastest in a maze and the second fastest on 2 of the maps tested. This would mean that for a game to incorporate the most believable algorithm it would have to sacrifice some of the efficiency of the pathfinding. However, this result may not hold true if the maps became larger or more open as they may favour heuristic based searches.

Recommendations

The limitations of the maps hindered the final conclusions of the testing. This could be enhanced by using a wider range of maps from different games, including larger more open maps and some smaller and condensed. The inclusion of a wider map pool would allow the algorithms tested to be equated to maps they prefer and some in which they are not optimised for. Larger selections of maps would also give participants diversity over the layout of the maps and the obstacles contained within. An additional enhancement of this would also to randomly generate the start and the end point to explore how the believability and navigation changes with distance.

Scaling of the testing is another area that could be improved, under the current method the heatmaps were collected and manually transferred onto each algorithm. However, to solidify the results of the testing, a larger group of participants would be ideal, scaling the current method would require an abundance of time and resources delaying other aspects of the research. Searching for alternative methods to complete the participants heatmap should be explored or tool created to correlate the results together.

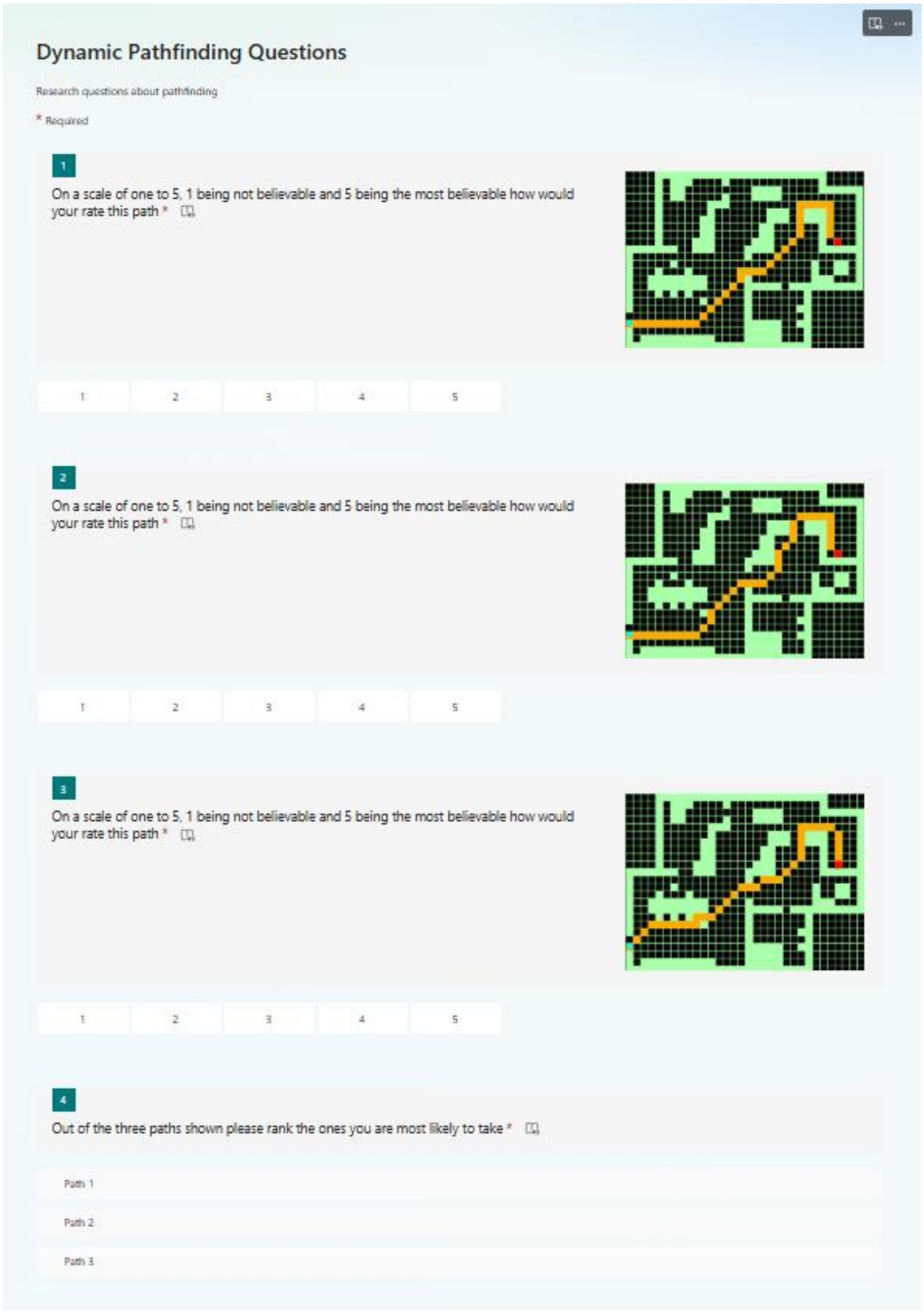
Reactivity underpinned a lot of the research and how an agent would navigate around dynamic obstacles, this relates very heavily to believability in many games. However, due to the nature of the study while D* Lite and parts of the artefact can handle dynamically placed obstacles this was not tested for and distracted parts of the research. Nevertheless, it is an interesting avenue for research and as games strive towards increasing reactivity something that is needed. Including this in the testing would enhance the overall results, this would be feasible by allowing participants to interrupt an agent’s path by placing obstacles and asking for feedback based on how the agent handled the interruption.

Alternative pathfinding algorithms could be explored to enrich the testing. With the limited number of algorithms, a wider range of maps and testing would only cement one of the three included as the one to use without consideration many other algorithms that could shine under conditions met through test that have otherwise been discarded.

In pursuit of believability the research conducted covered a wide area of pathfinding, decision making including enhancements and adaptations that have been made over the years. The focus on believability underlined the aims and objectives and directed the artefacts goals while testing. Technical and complex pathfinding algorithms were understood, adapted and implemented into an artefact that is expandable to be augmented with any additional pathfinding algorithms for future work. Finally, a methodology was established that will allow for testing to be improved upon and explored in a wider environment.

Appendix

Appendix 1 Questionnaire submitted to participants



Dynamic Pathfinding Questions

Research questions about pathfinding

* Required

1

On a scale of one to 5, 1 being not believable and 5 being the most believable how would your rate this path *

1	2	3	4	5

2

On a scale of one to 5, 1 being not believable and 5 being the most believable how would your rate this path *

1	2	3	4	5

3

On a scale of one to 5, 1 being not believable and 5 being the most believable how would your rate this path *

1	2	3	4	5

4

Out of the three paths shown please rank the ones you are most likely to take *

Path 1

Path 2

Path 3

Bibliography

Airlangga, G., 2024. A comparative analysis of pathfinding algorithms in static environments: modifiedA*, PSO, and FLA. *Jurnal Mantik,* 7(4), pp. 3967-3976.

Barnouti, N. H., Al-Dabbagh, S. S. M. & Naser, M. A. S., 2016. Pathfinding in Strategy Games and Maze Solving Using A* Search Algorithm. *Journal of Computer and Communications,* 4(11).

Bell, E., Bryman, A. & Harvey, B., 2015. *Business Research Methods.* Oxford: Oxford University Press.

Bethesda Softworks, 1996. *The Elder Scrolls II: Daggerfall. [Game]* s.l.:Bethesda Sodtworks.

Botea, A. et al., 2013. Pathfinding in Games. *Artificial and Computational Intelligence in Games,* Volume 6, pp. 21-31.

Bourg, D. M. & Seamann, G., 2004. *AI for Game Developers.* 1st ed. s.l.:O'Reilly Media, Inc..

Brodén, A. & Bohlin, P., 2017. *Towards Real-Time NavMesh Generation Using GPU Accelerated Scene Voxelization,* s.l.: Digitala Vetenskapliga Arkivet.

Bull, R., 2024a. Squad Behaviour Using a Task Stack Approach for Call of Duty: Strike Team. In: P. Roberts, ed. *Game AI Uncovered.* s.l.:CRC Press, pp. 43-53.

Bull, R., 2024b. Using Voxels for Environmental Cues. In: P. Roberts, ed. *Game AI Uncovered.* s.l.:CRC Press, pp. 133-139.

Champandard, A. J. & Dunstan, P., 2013. The Behavior Tree Starter Kit. In: *Game AI Pro.* s.l.:CRC Press, pp. 73-91.

Condò, M., 2024. SB-GOAP Self-Balenced Goal-Oriented Action Planning. In: *Game AI Uncovered.* s.l.:CRC Press, pp. 175-184.

Creative Assembly, 2014. *Alien: isolation. [Game].* s.l.:Sega.

Cui, X. & Shi, H., 2010. A*-based Pathfinding in Modern Computer Games. Volume 11.

Dennaton Games, 2013. *Hotline Miami.* s.l.:Devolver Digital.

Dijkstra, E. W., 1959. A note on two problems in connexion with graphs. *Numerische Mathematik,* December.pp. 269-271.

Dill, K., 2013. What Is Game AI?. In: S. Rabin, ed. *Game AI Pro.* s.l.:CRC Press, pp. 3-9.

Dyckhoff, M., 2015. Ellie: Buddy AI in The Last of Us. In: S. Rabin, ed. *Game AI Pro 2.* s.l.:CRC Press, pp. 431-442.

Graue, C., 2016. Qualitative Data Analysis. *International Journal fo Sales, Retailing & Marketing,* pp. 5-14.

Green, D., 2024. Auto-Generating Navigation Link Data. In: R. Paul, ed. *Game AI Uncovered.* s.l.:CRC Press, pp. 107-120.

Guo, X. & Luo, X., 2018. *Global Path Search based on A* Algorithm.* s.l., Atlantis Press.

Hart, P. H., Nilsson, N. J. & Raphael, 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics,* 4(2), pp. 100-107.

id Software, 1993. *Doom, [Game].* s.l.: id Software.

Iskanda, U. A. S., Diah, N. M. & Ismail, M., 2020. *Identifying Artificial Intelligence Pathfinding Algorithms for Platformer Games.* Shah Alam, IEEE.

Isla, D., 2005. *GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI.* [Online] Available at: https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai

Ji, Q. & Gao, C., 2007. Simulating Crowd Evacuation with a Leader-Follower Model. *IJCSES International Journal of Computer Sciences and Engineering Systems,* 1(4), pp. 249-252.

Kirilenko, D., Andreychuk, A., Panov, A. & Yakovlev, K., 2023. *TransPath: Learning Heuristics for Grid-Based Pathfinding via Transformers.* s.l., Proceedings of the AAAI Conference on Artificial Intelligence.

Koenig, S. & Likhachev, M., 2001. *Incremental A*.* s.l., NeurIPS.

Koenig, S. & Likhachev, M., 2002. *D*Lite.* Edmonton, s.n.

Koenig, S., Likhachev, M. & Furcy, D., 2004. Lifelong Planning A∗. *Artificial Intelligence,* 155(1-2), pp. 93-146.

Lester, P., 2005. *GameDev.net.* [Online]
Available at: https://www.gamedev.net/tutorials/programming/artificial-intelligence/a-pathfinding-for-beginners-r2003/
[Accessed 28 11 2024].

Lim, L. Y., 1968. *A Pathfinding Algortihm for a Myopic Robot,* Pasadena: California Institute of Technology.

Liu, D., 2023. Research of the Path Finding Algorithm A* in Video Games. *Highlights in Science, Engineering and Technology,* Volume 39, pp. 763-768.

Madhav, S., 2018. *Game Programming in C++: Creating 3D Games.* First ed. Boston: Addison-Wesley Professional.

Mäntysalo, J., 2024. *Graph search-based pathfnding in dynamic environments,* s.l.: UNIVERSITY OF TURKU.

Millington, I., 2019. *AI for Games.* Third ed. s.l.:CRC Proess.

Murphy, R. R., 2000. *Introduction to AI Robotics.* First ed. s.l.:Bradford Books.

Namco, 1980. *Pac-Man, Arcade [Game].* 1980: Midway Games.

qiao, 2023. *Pathfinding.js.* [Online] Available at: https://qiao.github.io/PathFinding.js/visual/ [Accessed 7 November 2024].

Rabin, S., 2017. The Illusion of Intelligence. In: S. Rabin, ed. *Game AI Pro 3.* 1st ed. New York: A K Peters/CRC Press, pp. 3-9.

Rabin, S. & Sturtevant, N. R., 2013. Pathfinding Architecture Optimizations. In: S. Rabin, ed. *Game AI Pro.* s.l.:CRC Press, pp. 241-252.

Rafiq, A., Kadir, T. A. A. & Ihsan, S. M., 2020. *Pathfinding Algorithms in Game Development.* s.l., IOP Conference Series: Materials Science and Engineering.

Rahmani, V. & Pelechano, N., 2021. Towardsahuman-likeapproachtopathfinding. *Computers & Graphics,* Volume 102, pp. 164-174.

Rajesh, V. & Wu, C. Q., n/d. *An Extension of Pathfinding Algorithms for Randomly Determined Speeds,* s.l.: s.n.

Rallabandi, S. & Roberts, P., 2024. Reactive Behaviour Trees. In: P. Roberts, ed. *Game AI Unvovered.* s.l.:CRC Press, pp. 54-63.

Reeves, M. C., 2019. *AN ANALYSIS OF PATH PLANNING ALGORITHMS FOCUSING ON A* AND D* [Master's thesis, University of Dayton],* s.l.: OhioLINK Electronic Theses and Dissertations Center.

Rivera, N., Hernández, C., Hormazábal, N. & Baier, J. A., 2020. The 2k Neighborhoods for Grid Path Planning. *Journal of Artificial Intelligence Research,* Volume 67, pp. 81-113.

Robert, P., 2024a. *Game AI Uncovered.* 1st ed. Boca Raton: CRC Press.

Roberts, P., 2022. *Artificial Intelligence in Games.* First ed. s.l.:CRC Press.

Roberts, P., 2024b. Believable Routes A Pathing Acceptability Metric. In: P. Roberts, ed. *Game AI Uncovered.* s.l.:CRC Press, pp. 73-82.

Schoener, E. R., 2024. *A Comprehensive Review and Practical Applications of,* s.l.: s.n.

Soutter, A. R. B. & Hitchens, M., 2016. The relationship between character identification and flow state within video games. *Computers in Human Behaviour,* Volume 55, pp. 1030-1038.

Stentz, A., 1994. *Optimal and efficient path planning for partially-known environments*. San Diego, IEEE.

Tatio, 1978. *Space Invaders. Arcade [Game].* Japan: s.n.

Thompson, T., 2017. *The Perfect Organism: The AI of Alien: Isolation*. [Online]
Available at: https://www.gamedeveloper.com/design/the-perfect-organism-the-ai-of-alien-isolation
[Accessed 24 October 2024].

Thompson, T., 2024. The Changing Landscape of AI for Game Development. In: P. Roberts, ed. *Game AI Uncovered*. 1st ed. Boca Raton: CRC Press, pp. 1-11.

Uzoeghelu, J. E., 2021. *ROBOTPATHPLANNING:EXPLORINGD*(STAR)LITE ,* Cyprus: NEAR EAST UNIVERSITY.

Valve South, 2008. *Left 4 Dead [Game].* s.l.:Valve.

van Toll, W. et al., 2016. *A comparative study of navigation meshes*. Burlingame, Association for Computing Machinery, pp. 91-100.

Wooden, D. T., 2006. *Graph-based Path Planning for Mobile Robots,* s.l.: School of Electrical and Computer Engineering Georgia Institute of Technology.

Wooldridge, D., 2024. Infinite Axis Utility System and Hierarchical State Machine. In: P. Roberts, ed. *Game AI Pro*. s.l.:CRC Press, pp. 168-176.

Xu, X., 2023. *Pathfinding js*. [Online]
Available at: https://qiao.github.io/PathFinding.js/visual/

Zarembo, I. & Kodors, S., 2013. *Pathfinding Algorithm Efficiency Analysis in 2D Grid*. s.l., s.n., pp. 46-50.