Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

Computer Games Development
Project Report Document
Year IV

Donal Howe
C00249662
Supervisor:
Oisin Cawley
26/04/2023

DECLARATION

Work submitted for assessment which does not include this declaration will not be assessed.

- I declare that all material in this submission e.g. thesis/essay/project/assignment is entirely my/our own work except where duly acknowledged.
- I have cited the sources of all quotations, paraphrases, summaries of information, tables, diagrams or other material; including software and other electronic media in which intellectual property rights may reside.
- I have provided a complete bibliography of all works and sources used in the preparation of this submission.
- I understand that failure to comply with the Institute's regulations governing plagiarism constitutes a serious offence.

Student Name: (Printed) Donal Howe

Student Number(s): C00249662

Signature(s): Donal Howe

Date: 24/04/2023

Please note:

a) * Individual declaration is required by each student for joint projects.

b) Where projects are submitted electronically, students are required to submit their student number.

c) The Institute regulations on plagiarism are set out in Section 10 of Examination and Assessment Regulations published each year in the Student Handbook.

Table Of Contents

List Of Figures

List Of Tables

Acknowledgements

I would like to thank my project supervisor Oisin Cawley for his assistance on this research project throughout the year which is greatly appreciated. I would also like to thank my classmates for making the course throughout the four years that much more enjoyable.

1 Project Abstract

The idea behind my research project is to question when compared with the dynamic pathfinding algorithm known as “Dstar Lite” under a games development context will other heuristic and non-heuristic pathfinding algorithms be more beneficial when implemented into someone’s game or should they rather use one of the several different pathfinding algorithms which has been implemented into the application, which contains the following algorithms : “Lifelong Planning Astar” which is an incremental heuristic version of the Astar pathfinding algorithm which allows for the replanning of the most optimal path without having to recalculate the entirety of the path, the “Astar search algorithm” itself, “Dijkstra’s search algorithm”, the “Depth First Search pathfinding algorithm” which is the only non-guided pathfinding algorithm in this application and the final pathfinding algorithm which is being compared inside of this paper being “Jump Point Search” which is a further extension on the Astar pathfinding algorithm and of course Dstar Lite itself being an incremental algorithm heuristic pathfinding algorithm which computes the shortest path from two given points on a grid and allows for the replanning of the given path without having to recalculate the path from start.

There is also the question on where these algorithms may or may not be applicable inside of different games as certain things may vary which include the following different edge weights(cost of travelling from cell to cell) the number of obstacles on a path and how they handle this and would it affect the time until the paths completion as well as the grids size which may not be the same across a game world and could vary depending on the type of game which is being made.

In this paper the applicable data has been put forward for each scenario and concludes whether the algorithms chosen when put against the Dstar Lite algorithm are a better alternative than the dynamic pathfinding algorithm or perhaps it may be the case where Dstar Lite is the more applicable algorithm for the scenario presented in this paper.

Not only will this paper provide the times that it took for these algorithms to traverse the graph and to find the shortest path under these scenarios, but it will also talk about the implementation itself and the degree of difficulty that was involved in the implementation of each algorithm and how it can influence the decision of the user to implement each algorithm into their game. This will be discussed by giving pros and cons to the implementation of each algorithm.

These algorithms will be given equal precedence when being compared to Dstar Lite and as such will be focused on equally so the reader can come to an informed decision on which algorithm that they want to implement into their game.

2 Project Introduction

In computer games development, developers may be faced with a problem with how to get their character from point a to point b. When they are faced with this problem they may come to the decision to implement a pathfinding algorithm of their choosing which best suits their games environment whether it be dynamic or static and that will safely get their character or game object from start to finish on a grid.

This is a problem core to gaming. In their process to trying to find a solution to this problem they may potentially implement several different pathfinding algorithms into their game world in order to try and find which one suited their game and its environment, which could potentially be quite time inducive and what my research project hopes to achieve is to compare several of these different pathfinding algorithms under different scenarios so that the reader can come to a decision without having to do separate implementations.

Where this problem becomes difficult to solve depends on what kind of grid they have implemented into their games and what is meant by this it is not whether the grid is one dimensional or two dimensional. It is meant that depending on what size of grid they have chosen to implement, whether the world is changing dynamically and how this will affect the grid.

This in turn would affect the times it takes for these algorithms to complete along with an increase in their memory usage and finally how they handle these changes. This may for instance destroy a Cell on the grid by placing a wall or some sort of obstacle on it and as such they will have to change the course of their path for getting to the chosen end point safely and as quickly as possible.

This is why the topic was chosen, by making comparisons between the several chosen heuristic and non-heuristic algorithms against the Dstar Lite search algorithm on dynamic and static grid. How the data which was collected as a result of the research may have an impact on developers and influence the games which they are or could be developing due to their lack of knowledge surrounding search algorithms.

Or perhaps that the algorithms that they have already implemented may perhaps shine in different scenarios to others. This can be achieved as in this project it does allows for a visual comparison between algorithms. This means that the user will be able to see the algorithms of their choice(once at a time) race against Dstar Lite in real time and the path which they take. This will perhaps allow the user to see visual differences in the path or perhaps it they are identical So not only will they have a visual representation of the speed in which these algorithms work they will also have data displayed inside of this paper to back up the decision that they may eventually come to.

The end goal of this research topic is to comprehensively and conclusively come to the most optimal decision for the reader of this project so that then in turn they will be able to go and implement the algorithm most suited to their problem and that which will hopefully in turn optimise their games speed and alongside that be able to understand each algorithm in such detail that they won’t have any problem explaining it to others either and be able to implement it into their own game.

3 Research Question

The question which this research project is built on is whether the Dstar Lite pathfinding algorithm is the best suited search algorithm for traversing a dynamic and static grid environments under a computer games development context?

4 Objectives involved in the making of this project.

This section of the paper presents the technologies and main objectives of this project which were used during the development of the project and the steps taken throughout the project until its completion.

4.1 Objectives

Come to a comprehensive conclusion on if the Dstar Lite search algorithm is better suited for use in games development than the other heuristic and non-heuristic search algorithms.

Compare the algorithms under a neutral environment to avoid any form of bias.

Compare the development of each algorithm under an equal basis where possible.

Compare the reactions of each algorithm to changes in the path.

Compare the time it takes for each algorithm to complete a search.

Compare how each algorithms time for completion reacts on different sized grid.

Compare the algorithms using depth or complexity of the code.

4.2 Technologies to be used:

SFML -2.5.1

Visual Studio 2022

C++

4.3 Steps to completion

1. Setup a dynamic 2D grid
2. Implement Dstar Lite search algorithms in c++
3. Implement “Lifelong planning Astar” search algorithm in c++
4. Implement “Astar” pathfinding search algorithm in c++
5. Implement “Dijkstra’s pathfinding” search algorithm in c++
6. Implement “Depth first search” pathfinding algorithm in c++
7. Implement “Jump Point Search” pathfinding algorithm in c++.
8. Create test environment to gather data of each algorithm.
9. Record the results collected.
10. Make comparisons of each algorithm against Dstar Lite

5 Literature Review

5.1 Background.

This section of the paper presents a review on the pathfinding algorithms those of which have been implemented into the application. These concepts and understandings discussed in this literature review are needed to make a comprehensive and complete comparison of these algorithms on a 2D game world grid under a computer games development context.

5.2 Pathfinding.

There are several things needed to be understood when it comes to pathfinding algorithms in games development. First thing is why are these pathfinding algorithms done in the first place? These algorithms are done with the intention of getting an ai character or object in a game from point a to point b in a game world.

What these algorithms do is generate a safe and potentially shortest path along the grid world for the character to traverse. This then allows for the characters movement more efficient than just manually moving them once they either hit into a wall or change their direction accordingly depending on where you want them to go.

There are Two main types of pathfinding algorithms which someone will come across when researching the topic and they are directed and non-directed algorithms. What is the difference between the two?

A non-directed pathfinding algorithm does not spend any resources on trying to figure out how far it is away from the end point, and rather it is simply moving blindly until it finds its selected destination.

A directed pathfinding algorithm does however spend resources on trying to figure out both how far it is away from the destination and also how far it is away from the start position. What they do is look around and access all the neighbouring nodes edge costs and as a result will then move to the one with the lowest cost. One way which the lowest value path is calculated is using heuristics. What is a heuristic?

A heuristic is used to affect an algorithms behaviour and guide them towards their chosen destination. It tells the algorithm an estimation of the cost of the distance of the node being evaluated from the destination node selected.

There are several diverse ways to calculate the heuristic value of a node such as diagonal, Manhattan but the one used in the application developed and what has been Implemented into the algorithms which we are investigating inside of this paper is a form of heuristic which uses Euclidean distance. Which will be explained later.

That is a brief explanation of what a pathfinding algorithm is, the several types and some of the behaviours which they show, that have been used as a part of my implementation. More key features which are needed for the readers understanding and will be explained in more detail throughout the paper.

5.3 Dstar Lite Search Pathfinding Algorithm

5.3.1 Overview

The Dstar Lite algorithm is an incremental heuristic pathfinding search algorithm which allows for the rapid replanning of a path after it has been found without the recalculation of the entire path.

How does it do this? This is achieved by retaining information from the previous searches. Rather than having to recalculate the entire path from scratch.

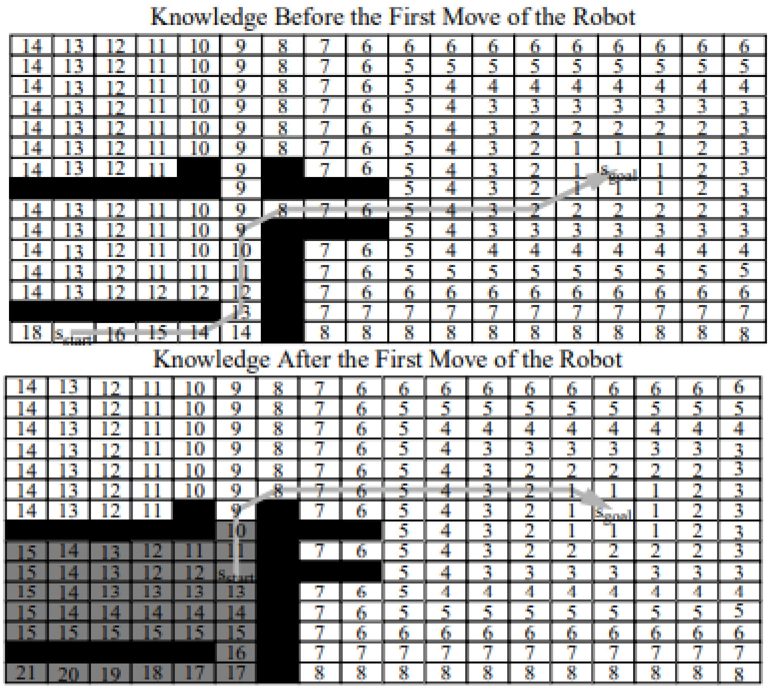

Figure 5-1 Dstar lite path.

(Koenig, S. and Likhachev, M. 2002)

5.3.2 Key Information

Key variables which need to be understood in order to implement the dstar lite algorithm. One thing to understand is that each node in the grid has these variables contained within them and as such each nodes values are separate to one another which in turn makes it easy for the priority queue to evaluate which node is to be expanded next.

“Gcost” – The Gcost of a Node is the distance from where that node(current point in the search) is on the given graph/grid to the start node.

“Hcost” – The Hcost of a Node is the distance from where that node (current point in the search) is on the graph/grid to the destination/goal node.

“Rhs cost” – The rhs cost otherwise known as the right-hand side value is used with a different understanding in dstar lite to the other places found elsewhere in robotics. In the context of dstar lite one can think of it as an estimation cost to the start node whereas the gcost is the actual cost to the start node. Keep that in mind for when this topic is discussed further throughout the paper.

“Key modifier”- The key modifier found in the application as “K_M” and is used as an offset for when the start position of the robot or character moves along the path to prevent more skewed cost values further up the path. How this affects our calculation of a variables key will be further explained soon.

“a nodes key” – a nodes key is a pair or in the application is an std::pair that holds two values calculated in the calculate key function.

(“aaai02b.pdf,” n.d.)

5.3.3 Algorithm

```
procedure CalculateKey(s)
return [min(g(s), rhs(s)) + h(s_start, s) + k_m; min(g(s), rhs(s))];
procedure Initialize()
U = ∅;
k_m = 0;
for all s ∈ S rhs(s) = g(s) = ∞;
rhs(s_goal) = 0;
U.Insert(s_goal, CalculateKey(s_goal));
procedure UpdateVertex(u)
if (u ≠ s_goal) rhs(u) = min_{s' ∈ Succ(u)}(c(u, s') + g(s'));
if (u ∈ U) U.Remove(u);
if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
while (U.TopKey() <˙ CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start))
  k_old = U.TopKey();
  u = U.Pop();
  if (k_old <˙ CalculateKey(u))
    U.Insert(u, CalculateKey(u));
  else if (g(u) > rhs(u))
    g(u) = rhs(u);
    for all s ∈ Pred(u) UpdateVertex(s);
  else
    g(u) = ∞;
    for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);
procedure Main()
s_last = s_start;
Initialize();
ComputeShortestPath();
while (s_start ≠ s_goal)
  /* if (g(s_start) = ∞) then there is no known path */
  s_start = arg min_{s' ∈ Succ(s_start)}(c(s_start, s') + g(s'));
  Move to s_start;
  Scan graph for changed edge costs;
  if any edge costs changed
    k_m = k_m + h(s_last, s_start);
    s_last = s_start;
    for all directed edges (u, v) with changed edge costs
      Update the edge cost c(u, v);
      UpdateVertex(u);
    ComputeShortestPath();
```

Figure 5-2 Dstar lite algorithm.

(Koenig, S. and Likhachev, M. 2002)

[1]How does the algorithm work? dstar lite is an extension of lifelong planning astar and is an incremental heuristic algorithm. Dstar lite rather than a typical search algorithm which searches from the start to the destination node, dstar lite does not do this it, but rather it searches backwards from the destination node to the goal node. Once an optimal path is found from destination to start the start position or where the robot is currently. The robot is then moved to the next viable node closest to the destination node. This is where our key modifier value comes into play for instance when the heuristic values are calculated initially based off the start

node, the robot has now moved and in turn our start node has changed, so we must increase the value of our key modifier. This is why it is easier to be explained as an offset for the change in robot position as otherwise, after the robot has moved if our key modifier was not implemented into the calculate key function the nodes heuristic value would not be correct once a recalculation has been made.

How is the shortest path found? we are dealing with two separate states of a node which is consistent and inconsistent nodes. What this means is that if a node is consistent, there is no need to make any changes to the nodes gcost values and rhs cost value as they are equal and in turn its values are consistent with one another.

Then there is the term known as inconsistent which means that a nodes g cost values and rhs values are not equal and as such we must deal with them to progress the search. Within this type of state there is two distinct types of inconsistencies. There is “over consistency” where our gcost is Greater than the rhs cost and then there is “under consistency” where our rhs value for the node is greater than the gcost value for the node.

How do we deal with this? First thing is to do make a check for over consistencies and once that is done we need to relax our gcost value down to our rhs value like what is done in Dijkstra’s Search algorithm this is done to make the gcost and rhs cost values equal to one another, once that is done we search the neighbours of the current node and assign the smallest rhs value of our neighbours to our current node and re-add the node to back into our priority queue to be investigated later. As seen above in Figure 5-2

In the case of under consistencies we need to assign our rhs value to infinity and update the node. The same thing is done where the rhs is calculated to and is returned into the queue with the updated value. Again this is done to make the node’s values consistent with one another. As seen above in Figure 5-2

However inside of the updating vertex function we need to check if that node is not already in the queue as to ensure we do not put that node into the queue if it is already there. With this information you can understand that the goal is to make nodes consistent with itself and will not be revaluated if the node is consistent. As seen above in Figure 5-2

How is the queue ordered? The queue is ordered using a functor which will return the node with the lowest cost. It does this by comparing a nodes key against the next node in the queue once it needs to be reordered.

What about in the case of nodes with equal values or a draw? The functor returns the node which is higher in the priority queue.

Once the best path is calculated how do we deal with changes? Dstar holds onto the path calculated on the earlier search and in turn uses this to update the path quickly as it does not need to recalculate the entirety of the path. What the algorithm does is it checks for any changes in the neighbours of the current robot position and if one of those nodes has an increase in their edge costs and are as such now not able to be traversed, we will recompute a best search around the node. As seen above in Figure 5-2

How to calculate the key of a node? To calculate the key of a node you do as such the first of the pair is the minimum value of the rhs cost and the gcost of the node plus the key modifier plus the heuristic value of that node. The second of the pair is the minimum of the Rhs value and Gcost value as seen above in Figure *5-2*.

With the individual aspects of dstar lite having been explained above the next question that needs to be asked is how does it behave? dstar lite behaves like what the dstar search algorithm behaves like with the exception that it is not as complicated to implement and does not have the same memory usage hence why it is coined dstar lite. When the path is initially calculated dstar lite acts like a greedy first search and what this means is that it takes the lowest costing node to the start node as remember dstar lite searches backwards from the destination node to the start node. One thing to note is that in an 8 directional graph where there is no extra cost for diagonal movement will lead to a zig zag pattern of movement for example:

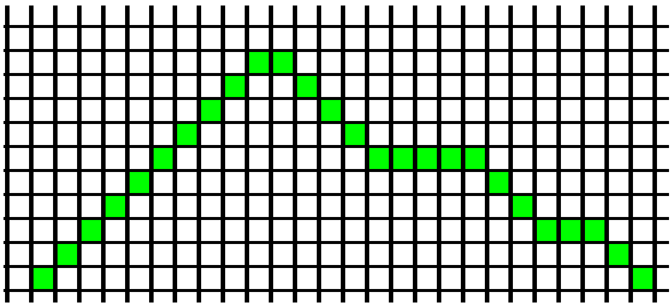

Figure 5-3 Dstar lite path on 2D grid.

To prevent this sort of movement one can simply add an added weighting to move diagonally. However, in my implementation this is not done as it was not meant in the overall design of the application. Next thing to take note of is how the algorithm acts once a node along the path becomes untraversable with a wall on the path. One can note how the wall has gcost and rhs cost of infinity.

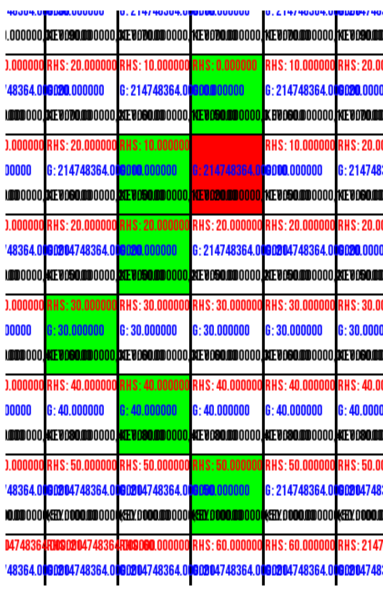

Figure 5-4 How Dstar path changes with wall on path.

That is an overview of the dstar lite search algorithm and how it works on a 2D dynamic grid aswell as how it behaves when a wall is placed on the given path (Koenig, S. and Likhachev, M. 2002)

5.4 Astar Search Pathfinding Algorithm.

5.4.1 Overview

Astar is a non-incremental heuristic search algorithm which means that is solves the traversing problem from scratch and that it knows the end and start point. It then tries to find the shortest path to the end point; However it will rerun itself if an obstacle gets in the way. Astar can find the shortest path through a priority queue which will compare the values of each node using both their Hcost(distance from the node) and Gcost(distance from the start node). This is how it knows to look at certain nodes first. Astar unlike Dstar Lite does not retain any information from search to search. Astar is a very widely used pathfinding algorithm in the games development industry.

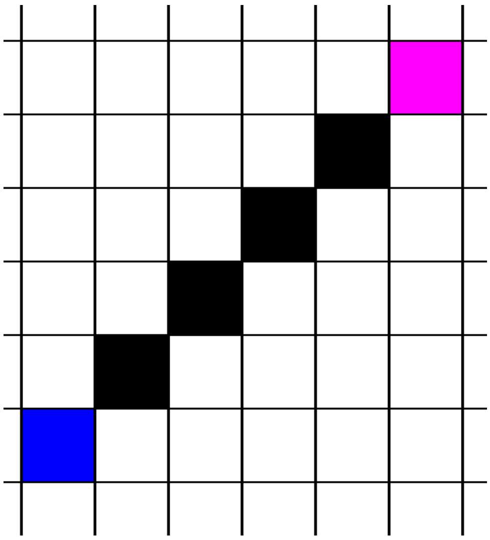

Figure 5-5Astar path.

5.4.2 Key Information

When understanding how the Astar pathfinding algorithm works one must first understand some key features which is used inside of Astar. Note the calculating of a nodes heuristic value is the same as in Dstar Lite there is not any additional costs for moving diagonally as previously mentioned one could be easily implemented.

“Fcost” – in Astar the Fcost is the value of the Gcost 0 plus the value of the Hcost (mentioned in 5.3.2). what this value does is allow us to easily investigate a certain node based on its location in the grid.

“Gcost” – (mentioned in (Koenig, S. and Likhachev, M. 2002)

Key Information 0)

“Hcost” – (mentioned in (Koenig, S. and Likhachev, M. 2002)

Key Information 0)

“weight” – weight is the cost it takes to move to a node. If you want to place a wall on the grid you can change the node which the wall resides in have a weight of infinity so that node will never be investigated.

“Previous pointer” - this is for constructing the path. This can also be known as the parent node of a given node and is used to reconstruct the path once it is known.

“Heuristic” – despite it being calculated in the same way as Dstar lite as mentioned above we must note how the heuristic can greatly affect the efficiency and behaviour of our Astar algorithm. If the heuristic is less than the cost of moving to the goal then it will always find the shortest path” If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then A* is guaranteed to find a shortest path. The lower h(n) is, the more node A* expands, making it slower. **(**Patel 2019)”, but if it is greater than the cost of moving to the goal then it may not find the shortest path “Ifh(n) is sometimes greater than the cost of moving from n to the goal, then A* is not guaranteed to find a shortest path, but it can run faster. **(**Patel 2019)”, as you can tell depending on how we write our heuristic you could potentially skew the time or path given back to us by Astar and as we want the fastest time possible and best path possible this is important to be sure about.

Note how Astar does not have an Rhs cost this is due to it not being an incremental algorithm and does not need to retain any information instead it completely recalculates the path.

5.4.3 Algorithm

The Astar algorithm works as such first thing you need to initialize all of the nodes inside of your grid’s values. Calculate their Hcost distance from the goal node as well as setting them gcost to infinity. Set their parent Cell to be a null pointer as this will be assigned later in the function. Next you need to establish a priority queue which takes a functor that will compare the fcosts of a node to one another or just what their fcost value would be.

Once this has been done you need to insert the start cell into the priority queue and set its Gcost to 0 and it to have been marked/visited and search all their neighbours(it will go in order of the lowest Fcost value due to the functor). If the node is not equal to its parent it will check to see if the distance to the child(child cell/ current cell’s Gcost + the weight it takes to move there) is less than the Gcost of the child. Then you will set that current node parent to be the top of the priority queue and its Gcost to be the cell at the top of the priority queue’s Gcost + its weight. Then if the current node which you are searching through is the goal node terminate the search. Otherwise if this is not the case and the nodes weighting and Gcost is not less than the child’s Gcost then simply set that current cell as marked and pus it to the queue. If the current cell runs out of neighbours to be searched remove it from the queue when done the algorithm should look like the below **Error! Reference source not found.**.

This is an overall view of the Astar search algorithm and all the key information in which you need to know before this paper can correctly display my findings and make comparisons between Astar and Dstar Lite (Swift, N. 2020)

5.5 Lifelong planning Astar Search Pathfinding Algorithm.

5.5.1 Overview

Lifelong planning Astar is an incremental heuristic pathfinding algorithm that finds the path by updating the values of nodes from previous searches rather than recalculating the entire graph and having to start from nothing it is one step down from Dstar Lite which is a continuation of the lifelong planning astar algorithm. The algorithm uses the heuristic function to guide itself towards the goal node. Unlike dstar lite lifelong planning astar searched from the start node to the goal node. When dealing with changes to the grid it re-expands nodes from previous searches which have been affected by the change and their predecessors to replan the path

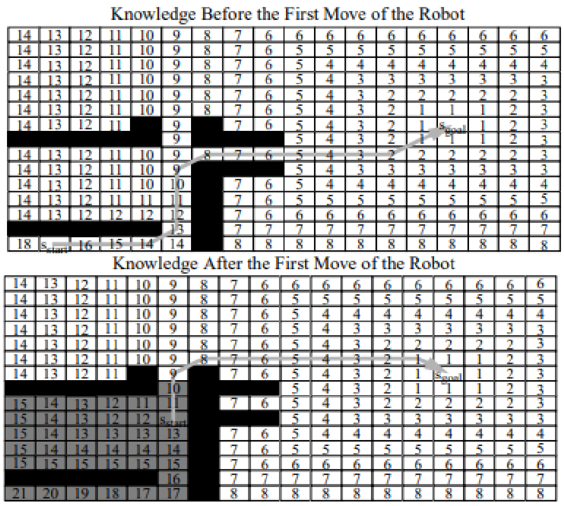

Figure 5-6 Lpa* path.

(Koenig, S. and Likhachev, M. 2002)

5.5.2 Key Information

To get a better understanding of lifelong planning astar one must first understand the key information which is listed below:

“Closed List” – this is a list of nodes which have already been expanded by the algorithm.

“Open List” – this is the priority of nodes which is currently being expanded by the algorithm.

“Rhs cost” – (mentioned in (Koenig, S. and Likhachev, M. 2002)

Key Information 0)

“Gcost” – (mentioned in (Koenig, S. and Likhachev, M. 2002)

Key Information 0)

“Key Value” – (mentioned in (Koenig, S. and Likhachev, M. 2002)

Key Information 0), the key value is calculated the same way to dstar lite with the exception there is no key modifier involved with the calculation.

“Start Node” – this is the start node of the algorithm, i.e. where it starts.

“Goal Node” – this is the goal or destination node i.e. where you want to get to.[2]

That is all the key variable information which you need to understand the functionality of Lifelong planning Astar.

5.5.3 Algorithm

```
procedure CalculateKey(s)
return [min(g(s), rhs(s)) + h(s, s_goal); min(g(s), rhs(s))];

procedure Initialize()
U = ∅;
for all s ∈ S rhs(s) = g(s) = ∞;
rhs(s_start) = 0;
U.Insert(s_start, CalculateKey(s_start));

procedure UpdateVertex(u)
if (u ≠ s_start) rhs(u) = min_{s'∈Pred(u)}(g(s') + c(s', u));
if (u ∈ U) U.Remove(u);
if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));

procedure ComputeShortestPath()
while (U.TopKey() <̇ CalculateKey(s_goal) OR rhs(s_goal) ≠ g(s_goal))
  u = U.Pop();
  if (g(u) > rhs(u))
    g(u) = rhs(u);
    for all s ∈ Succ(u) UpdateVertex(s);
  else
    g(u) = ∞;
    for all s ∈ Succ(u) ∪ {u} UpdateVertex(s);

procedure Main()
Initialize();
forever
  ComputeShortestPath();
  Wait for changes in edge costs;
  for all directed edges (u, v) with changed edge costs
    Update the edge cost c(u, v);
    UpdateVertex(v);
```

Figure 5-7 Lpa* algorithm

(Koenig, S. and Likhachev, M. 2002)

The calculation of the key In Lpa* works as such, it is an std::pair and the first of the pair is the minimum cost between the Gcost and the Rhs cost + the heuristic value from the goal node. The second of the pair is the minimum of the G cost and the Rhs cost.

If the node is over consistent as explained in (Figure 5-3) relax it down to its rhs value and add all the neighbours of that node to the open list. If the node is under consistent as explained in (Figure 5-4) update that nodes g cost to infinity and add that nodes neighbours to the open list. If the node has already been expanded terminate the current search. If this is not the case update the nodes rhs value and add it to the closed list. For each node that is in the closed list and its neighbours we want to update their rhs values if they go through nodes which are in the process of being expanded. Then lastly for each node in the open list we want to expand each of their neighbours and update their key values with the new rhs cost and g cost. As the way to deal with nodes being both over and under consistent having been already explained in the explanation of how Dstar Lite works it will not be replicated here. However it is the same way of dealing with them in this case. (Koenig, S., Likhachev, M. and Furcy, D. 2004)

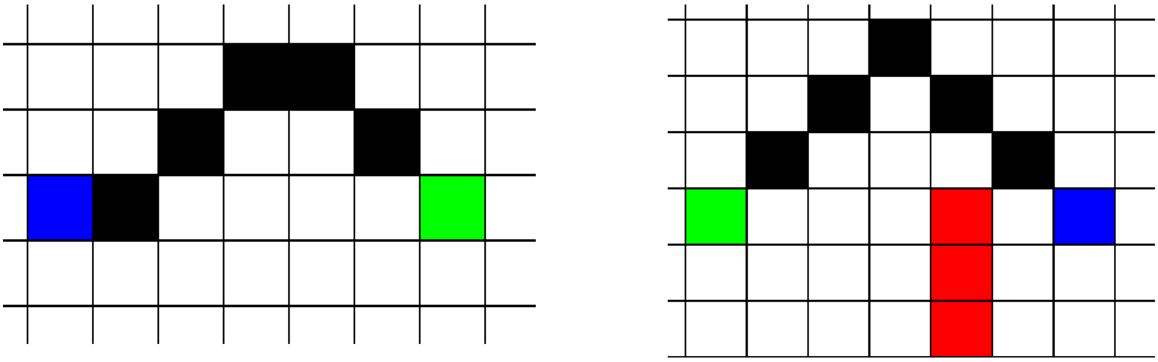

Figure 5-8 Lpa* path without walls vs with walls.

5.6 Dijkstra’s Search Pathfinding Algorithm

5.6.1 Overview

Dijkstra’s search algorithm is a guided search algorithm that uses node weights and connections to find the shortest path to the goal node. Whereas Astar uses the heuristic value distance from the goal node as hcost and distance from the start node Gcost to find the path, Dijkstra’s only uses the distance from the start node of each node however may not find the shortest path to the goal node which is why it is not considered to be as good as Astar however we are comparing it to Dstar Lite so it could potentially be more beneficial under a games context to the developer.

5.6.2 Key Information

“Relaxing” – what this is we relax the cost value down from infinity to the actual value from the start node.

“Gcost” – as mentioned in (5.3.2)

“Source/goal Node” – this is the node we want to find the shortest path from all other nodes on the graph to

“Start node” – as mentioned in (5.3.2)

“Previous pointer” – as mentioned in (5.4.2)

“weight” – as mentioned in (5.4.2)

5.6.3 Algorithm

One thing to note when it comes to the Dijkstra’s algorithm it works similarly to the Astar algorithm in that it’s a greedy first search by this it will organise the priority queue based on the lowest G cost of a cell as Dijkstra’s does not use a heuristic function and as a result its hcost is set to zero and its neighbours then investigating the cell with the lowest cost. The priority queue is organised using a functor like Astar but rather than using the addition of a cells hcost an gcost it only compares the cells based on their g cost.

The algorithm will relax down a cells Gcost from infinity down to the actual cost from the source node and as a result it can be guided towards the goal node which you have chosen.

When you want to reconstruct the path you can do the same as Astar and set the previous pointer of that cell to the parent of it if its gcost value is less than of its parent and is a part of the shortest path. Then for reconstructing the path you simply just need to loop back through the pointers from the destination cell. An example of a path with and without a wall using the Dijkstra's search algorithm is shown in (Figure 5-21 Dijkstra's path without wall) and (Figure 5-21 Dijkstra's path with wall on path)

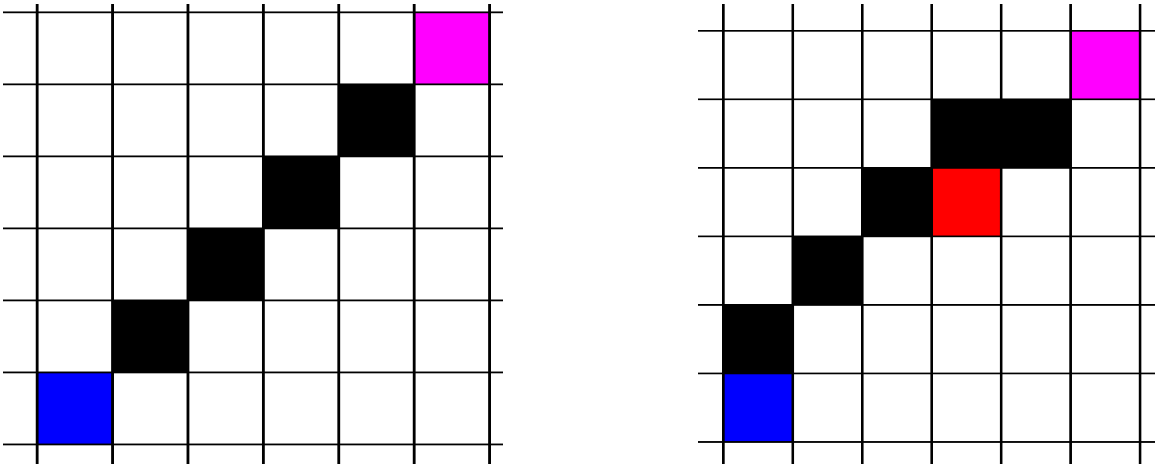

Figure 5-9 Dijkstra's path without walls vs with walls.

The final thing to note is that when comparing the Dijkstra's search algorithm to dstar lite is that this algorithm is less efficient than astar certain search conditions so how it will compare to dstar lite is remarkably interesting. But overall your understanding of Dijkstra's search algorithm should be inepter to have a better understanding of the data which has been collected as a result of the commenced research. (Rachmawati, D. and Gustin, L. 2020)

5.7 Depth first Search Pathfinding Algorithm

5.7.1 Overview

As previously discussed early in the paper of the two diverse types of pathfinding algorithms being directed and non-directed algorithms, depth first search is an example of a non-directed search algorithm. It does not have a heuristic function to guide it in any way. How depth first search works is rather than using a heuristic function to guide it towards the goal node it simply picks a direction in which it wants to search, and it will go in that direction until it no longer can and then it will pick a new direction to search, and this process is recursive until the goal node is finally found. Where the version of the typical depth first search algorithm differ is that the version inside of the application is capable of not searching untraversable nodes/ nodes where there is a wall or obstacle.

5.7.2 Key Information

"Neighbours List "- this is the surrounding neighbours of a given node.

"Recursive Function" – recursive function is one that will call upon itself again inside of the function.

" Previous pointer" as mentioned in (5.4.2) this is used for tracking the path taken.

5.7.3 Algorithm

In my description of the algorithm it will discuss more why the function is recursive but one thing to note is that compared to the other algorithms depth first search is missing data structures as there is no need for them. For example there is no use for a functor as we do not have any instance of comparison in the algorithm, nor do we need data structures such as priority queues as we don’t have to store the path in a data structure where we need to compare the cells against one another we can just simply use a stack once each node in the path has a parent cell appointed to it.

As previously explained since this algorithm uses the concept of a recursive function you have no need to use any data structure. When you want to store the path you can simply just track your way back through the path using the parent pointer/ previous pointer as mentioned previously to store the path.

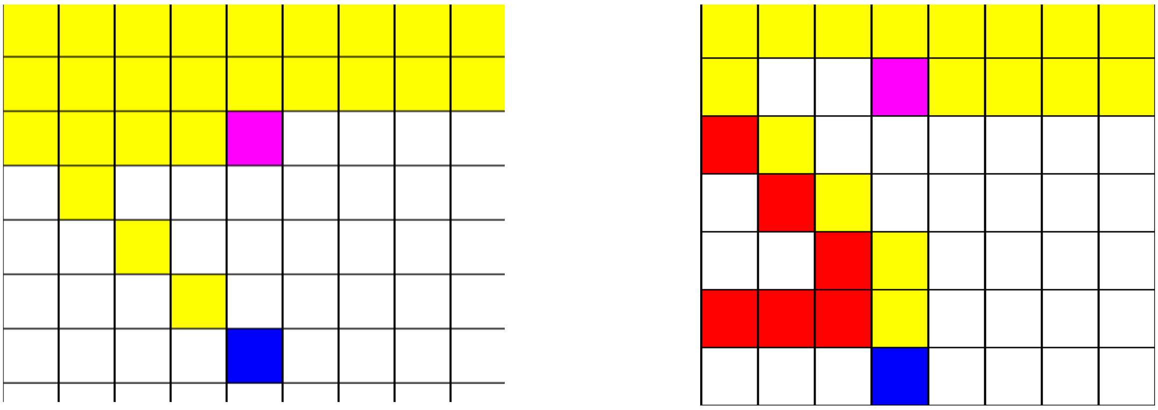

Figure 5-10 Depth first search algorithm without walls vs with walls.

The final thing to note is that when comparing the depth first search algorithm to dstar lite is that it is very different to it, so when the topic was researched, it was an important question to have and see how it will hold up against the dynamic search algorithm will it be more beneficial for developers to implement dstar lite into their game or use depth first search. (Kaur, N. and Garg, D. 2012)

5.8 How to compare the algorithms.

When this topic was decided on, and production of the application had commenced including the selection of the appropriate algorithms to compare were set in stone the next thing that had to be decided was how to compare them. There is the obvious way in which to compare them which is simply by time taken to find the goal node however this is not necessarily the fairest way to compare them as Dstar Lite is going to take longer as it does more calculations and holds onto more memory, and you won't see its benefits until you make a change to the path. So here are the ways in which have been selected to compare these algorithms:

1. Performance- speed of the algorithm and the memory usage, this value of measurement may be the most important form of comparison when it comes to games as speed and memory usage is vital for games development.
2. Optimality – comparing how often each algorithm returns to the best path, this is quite difficult for a direct comparison as not every algorithm check for ties in terms of quality of path however is still extremely vital for comparisons sake.
3. Robustness – speed with obstacles on the path, this will also be accounted for when after the algorithm has completed its search as mentioned above if the path changes how does the time get affect and how much of a detriment occurs to the recorded time.
4. Scalability – for increase and decrease of the grid sizes, in the case of grid size the algorithms were each compared on three separate grid sizes ranging from a 10x10 grid size to a 100x100 grid size to get more accurate results when the path is changed this will affect Astar and the others more greatly to Dstar Lite.
5. Implementation – how difficult each algorithm is to implement; this is important as the difficulty in which it was to implement these algorithms may change your decision and sway you to choose another algorithm to implement.
6. Depth of code i.e., how many for nested for loops or conditionals that increase the complexity of the code, this may be a degree in which the code is more likely to break and maintain for the developers which also could affect the decision of the reader and my final evaluation on whether you should use Dstar Lite in a game's development context. (Pathak, M.J., Rami, S.P. and Patel, R.L. 2018)

5.9 Controls necessary for fair comparison

The next thing that needs to be decided on is a list of controls for the testing and comparisons and how they will be implemented into my application. This is extremely important as in my research of these algorithms the last thing that was wanted was to provide the possibility of bias or an advantage to one algorithm or the other as this would skew the integrity of the results which have been collected because of my investigation into these algorithms.

1. Map configuration, i.e., sizes should be the same and the same number of obstacles should be placed for each user, each algorithm operated on the same grid size in each test to avoid skewed results.
2. Same heuristic function they should be the same which they are.

3. Same termination conditions i.e., after a certain amount of time and iterations done This is the same for each algorithm it is either after the goal is found or if enough time has passed.
4. Implementation details – use the same data structures i.e., vectors and priority queues etc., the algorithms use the same data structures where applicable, for instance Depth First Search has fewer necessary data structures to the rest of the algorithms.
5. Statistical analysis to evaluate speed.
6. Randomization – random start and end positions for testing to ensure no bias, this was done in the collection of data as there is a separate environment with this capability inside of the application called testing which does not allow the user to pick the start and end points of the algorithm along with the number of walls placed on the path.
7. Number of trials for the data, each algorithm must have the same number of trials in each given test as one another to avoid bias or potential average time/ average memory usage calculation error.
8. Path length – static start and end pos are the same distance away from each other when gathering the results.

6 Study/Methodology

This section of the paper shows how theories of pathfinding algorithms researched in section 5 were implemented in this study.

6.1 Implementing Astar

When implementing Astar, having gotten the knowledge from the research that had been done the first thing to do was to create the functor which would compare each cell Fcost to one another and return the one with the smallest value.

The first step was to ensure that each cell in the grid had the necessary variables inside of them such as their “Hcost”,” gcost” and “fcost” these variables were explained inside of the literature review.

The next step of the implementation was to setup the functor which would compare the two cells fcost value and return the lower of the two.

The initialise astar function simply assigns all of the correct values for each cell in the grid for the search to commence.

The compute shortest path follows the Astar search algorithm to find the shortest path to the goal node by calculating the values for each cell as it progresses through the path. It then also assigns the parent cell or previous cell to the current cell being investigated. It does this to make it easy to reconstruct the path back from the goal node to the start node. It then returns the path inside of a stack which is organised by last in first out.

This algorithm was then tested on the different paths with no obstacles and changes to the path mid search and also in turn on paths with obstacles and changes to the path mid search. Astar has to recalculate the path if any obstructions occur mid search which is important to note. It was also tested on the three different grid sizes with the same types of trials being implemented on those grid sizes.

6.2 Implementing Dstar Lite

When the implementation of Dstar Lite had begun and all of the algorithms had been understood completely and it was possible to successfully implement the Dstar Lite search algorithm in full it was implemented in this process

The first thing was to make sure that the cells on the grid had the necessary variables for Dstar Lite to work such as an “rhs cost” value, “g cost” value and a key value, each of these variables which will be actively changed during dstar lite’s search through the grid.

The second step is to setup the functor for Dstar Lite what this does is it compares the cells key’s first values against one another and their second key values against one another again. If in the case of a tie it will return the higher in the priority queue

The third step is to create the calculate key function which sets the key values of a node. What the calculation is, is that it gets the minimum of that cells “gcost” and “rhscost” adds that value to the cells heuristic value as well as the key modifier. The second of value of the key is the minimum value of the “gcost” and the “rhscost”.

The next step was implementing the main function of the algorithm which has a purpose of moving the start node or can be considered as the characters position through the path, updating the key modifier, and then handling changes to the path. This also returns the final path.

The initialise Dstar Lite function is where you set the values of the grid which is necessary for the algorithm to work in here you will also set the key to the start node and push that start node into the priority queue.

The compute shortest path function works as such this Is where you dictate how the algorithm handles the different types of inconsistencies mention above in the literature review. This will either relax down the gcost of the cell being investigated or it will raise the rhs cost to the gcost.

The next and final function that needs to be implemented is the update vertex/node/cell function this will assign the lowest value rhs cost of that cell's neighbours to that cell. It then checks to see if that cell is in the priority queue if it is in the queue it will take that cell out of the queue it will then go and recalculate that cells key values with the new rhs cost and will once that be complete push the cell back into the queue with the updated values to be potentially investigated in the future of the search.

When having implemented Dstar Lite these were the steps taken as a result of the research that had been done. This algorithm was then tested on paths with no obstructions and changes to the path and paths with obstructions and changes to the path. It was also tested on the three different grid sizes with the same types of trials being implemented on those grid sizes.

6.3 Implementing Dijkstra's Search Algorithm

When implementing Dijkstra's search algorithm based off of the research gathered there was a few steps which had to be implemented in order for a correct implementation of the algorithm.

The functor which was implemented only compares the gcost values of each cell against each other.

The compute shortest path algorithm finds the shortest path by sorting the cells inside of the priority queue by their gcost values. It does this until it finds the goal node. Like Astar a parent cell is set for an easy reconstruction of the path after the goal node is found.

This algorithm was tested under the same conditions as the rest as mentioned in the literature review.

6.4 Implementing Lifelong Planning Astar

When the implementation of Lifelong Planning Astar had begun, and all of the algorithms had been understood completely and it was possible to successfully implement the Lifelong Planning Astar search algorithm in full it was implemented in this process.

First thing was to setup the functor which works the same as Dstar lite it compares the cells based off of their key values it then handles ties by returning the higher of the two in the priority queue.

The compute shortest path does works in the same regards as dstar lite as it handles the inconstancies it also then passes the cells into the update node function which gets the minimum value and recalculates the key and puts it back into the priority queue.

This algorithm was tested under the same conditions as the rest of the algorithms in order to endure a fair evaluation of each algorithm.

6.5 Implementing Depth First Search

Depth first search was implemented using only one function which works recursively until the goal node is found it follows the direction it is going until it cannot go in that direction any longer it also assigns a parent cell so the path can be reconstructed with ease.

This algorithm was tested under the same conditions as the rest of the algorithms in order to endure a fair evaluation of each algorithm.

7 Evaluation and Discussion

This section of the paper demonstrates the results gathered from the trials performed as a result of the algorithms being run on a two-dimensional grid each time was recorded as seconds and as such their average times will reflect this decision.

7.1 How will the data be displayed?

When comparing the times that it took for each search algorithm to find the destination/ goal node, it must first be noted how each algorithm had their times stored for each trial inside of seperate excel files for each algorithm.

So as a result each algorithm has three seperate excel files with each file containing their indivdual times when being run on each of the different sized grids for example the astar algorithm has the three seperate excel files called “AstarTime.csv” for the small sized grid, “AstarTimeMedium.csv” for the medium sized grid and then “ AstarTimeLarge.csv “ for the large sized grid.

Within these excel files are the times stored under the following criteria, basic path with no obstacles, one wall on the path, as well as two, three and four walls on the path. For this experiment to be successful and avoid any form of positional or any other froms of bias potentially given favorable advantage to any of the chosen algorithms it was necessary for the use of a random position on the grid however the number of walls on the path were kept the same for each algorithm.

To correctly display the time an average over a variety of trials under each the different types of scenrios for the algorithms was collected and calculated. For collecting the data that was used to calculate the average times on each under each scenario the first fifty trials were gathered as certain algorithms were tested in an uneven number of times for developmentive purposes. For example it took longer to implement Dstar Lite than it did the astar search algorithm so the tests for Dstar Lite were not under any specific criteria to ensure that the algorithm was working as intended.

These times have not been proccessed in the data collecetion. This is an example of how the average times were calculated. It’s the total time it took for every single trial divided by the total number of trials that were executed to get the average times each algorithm was executed the same number of times during the testing of these algorithms, the algorithms were then ranked from best to worst on the tables below.

8 Results

8.1 Calculation of average times examples

8.1.1 Small grid times with 0 Walls

Algorithm	Average Time (Seconds)	Ranking Based on Quickest Average Time (Seconds)
Astar	0.00090312	1
Dstar	0.06404	4
Dijkstra's	0.0043366	2
Lifelong planning Astar	0.0463567435897436	3
Depth First Search	0.3456.	**6**

Table 8-1 Small grid times 0 walls

8.1.2 Medium grid times with 0 Walls

Algorithm	Average Time (Seconds)	Ranking Based on Quickest Average Time (Seconds)
Astar	0.053890	2
Dstar	0.4816679024	5
Dijkstra's	0.03953446	1
Lifelong planning Astar	0.06336868	3
Depth First Search	0.197808.	4

Table 8-2 Medium grid times 0 walls

8.1.3 Large grid times with 0 Walls

Algorithm	Average Time (Seconds)	Ranking Based on Quickest Average Time (Seconds)
Astar	0.18156930	1
Dstar	4.03639998	5
Dijkstra's	0.34433232	2
Lifelong planning Astar	1.3943158823529413	4`
Depth First Search	0.788583.	3

Table 8-3 Large grid times 0 walls

Grid Type	Best Algorithm	Quickest Average Time (Seconds)	Fastest algorithm factors faster than slowest average algorithm time
Small grid size best average time	Astar	0.00090312	7.09 times faster than Dstar Lite
Small grid size worst average time	Dstar Lite	0.06404	
Medium gird size best average time	Dijkstra's	0.03953446	12.18 times faster than Dstar Lite
Medium grid size worst average time	Dstar Lite	0.4816679024	
Large grid size best average time	Astar	0.18156930	22.23 times faster than Dstar Lite
Large gird size worst average time	Dstar Lite	4.03639998	

Table 8-4 Comparison results.

7.1.4 Comparing the implemntation of the algorithms

The next type of comparison which needs to be discussed is how difficult it was to implement the algorithms when compared to the difficulty of implementing the Dstar Lite search algorithm this will be displayed in a table which will rank them from most difficult to least difficult.

Algorithm	Ranking
Dstar lite	6
Astar	3
Dijkstra's	2
Depth First Search	1
Lifelong Planning Astar	5
Jump point Search	4

Table 8-5 Comparison of implementation

With the ranking as seen above one can take from this that Dstar lite was the hardest algorithm to implement as it has the greatest number of features which need to be considered when implementing the algorithm as it is far more advanced programming than the likes of depth first search and Astar search for example.

9 Project Milestones

This section of the paper shows the project milestones throughout the year, it discusses the overall progress of the project and how it was delayed or if it was on schedule and the result of such hindrances and delays on the project.

When upon the commencement of this project the scope of the project was quite significantly smaller. For instance there was only two algorithms involved in the project and only two were going to be tested ,these two algorithms were the astar search algorithm and originally it was to be compared against the Dstar algorithm.

In the beginning, the project milestones were adhered to in the regards that the project had made efficient project with the documentation as well as the technical project. However, with further research into the "dstar search algorithm" and alternate versions such as "focused dstar" and "dstar lite", it was decided that in reference to information regarding the dstar lite search algorithm, it was and is more commonly used when it comes to dynamic pathfinding and dstar itself is not really used as much.

So the decision was made switch the dstar lite algorithm. This in essence was what the second draft of what the final project was going to be. Following this the decision to extend the research into the different pathfinding algorithms was made, the result of which drastically improved the understanding of these algorithms when it came to continuing the research, as a result the decision was made to compare dstar lite against more pathfinding algorithms.

The first of which that were decided upon was to compare dstar lite against Dijkstra's search algorithm then the next process was to compare it against a non-guided heuristic algorithm, so depth first search was chosen. This was the third draft of the project which was now in motion.

After the Christmas break work on the project had a delay on it due to some unforeseen circumstances such as covid-19 and of course college coursework. Finally the decision was made to compare dstar lite to another incremental pathfinding search algorithm, so the decision was made to compare it to lifelong planning astar.

The next iteration of the project was the visual component which meant how was the project going to this component and show the project to non-developers and can they be able to discern the difference between these algorithms from only a visual component.

So the method was chosen, and it was to have the algorithms on two separate screens always being compared to dstar lite. The user can see three separate screens one which changes the size of grid and which algorithm they want to use. The second screen is the editable grid, what is meant by this is that they can place down walls, start and endpoints on this grid and they will see the algorithm which they had chosen to traverse the grid.

The third screen it the visual demonstration of dstar lite, what this screen does is that it copies all input from the user on screen two and copies it into its own grid. It then in turn runs dstar lite on this screen. They can also see a debug version of this screen which shows more in-depth information about the algorithm and the effects it has on the grid. This was the fourth and final edition of the project which you can see today. Those were the project milestones and iterations of the project, throughout the course of development the milestones in regard to due dates set by the lecturers were adhered to mostly but not all of the time due to the difficulty of understanding dstar lite and also implementing dstar lite. As prior to the project I had no knowledge of incremental pathfinding and not helped by the limited resources surrounding the topic.

10 Major Technical Achievements

This section of the paper lists the technical achievements made inside of the project.

What are your major technical achievements?

- The major technical achievements are but not limited to the following:
- The Implementation of Dstar Lite
- The Implementation of Lifelong planning astar
- The Implementation of Jump point search
- Having both paths appear to the user on the screen(can see chosen algorithm path and dstar lite algorithm on separate screen which is toggleable)

11 Project Review

This section of the paper discusses an overall review of the project including what went well and if there were any problems throughout the making of the project. It gives an understanding on if the technologies used were inefficient and if there were any other better solutions available to be used if the project was to be built again.

What went right?

When researching this project most of the project went well for the most part as these algorithms is quite well documented and are easily understood so I was able to fully comprehend how these algorithms work so they could as a result be replicated, and accurate results were able to be gathered.

What went wrong was mostly the research into dstar and dstar lite as there are few resources surrounding these algorithms available. As a result it was quite difficult to research and gather a complete comprehension of the source material as these algorithms are not used in the games development industry but rather in robotics, so those papers are written through a robotics context. Why was this a problem? This is a problem as without external information and context on how these algorithms work it can be quite difficult to get a grasp on. This was the only thing really that hindered me or went wrong in the project although it finally got implemented this would be the only drawback.

As a result of this hindrance is that if I were to approach this topic again I would enter into my research with the understanding that there is little material surrounding the topic and none of the material is in a games context and that is how I would advise someone else to approach the topic as well.

regard to the technologies used using sfml and c++ was not as much a problem but there would be easier avenues to design the project as whole inside of a game's engine such as unity. With this in mind it is exceedingly difficult to make sfml projects including ones of this nature to look visually appealing and one has to go to greater lengths to do so whereas if it was done in a game's engine such as unity. Granted despite the drawbacks having to design every detail from the ground up with sfml and c++ lead me to a greater understanding of pathfinding algorithms and user interface design as a whole as I encountered more problems as a whole. Everything was of my own design and creating with no external packages is what I mean by this. For instance where I had to go and research the astar algorithm in unity this algorithm can be obtained through an external package the grid which I designed can be accessed through a nav mesh agent.

12 Conclusions and discussion of results

This chapter will discuss the results collected as a result of the research commenced throughout the year and conclude on which algorithm is best suited for computer games development after taking the results into account.

As the result of the data collected during the development of this project and compared having been compared to the data collected by the other algorithms it cannot be recommended that the developer implements the Dstar Lite search algorithm into their game and here is why.

From the data which has been put forward as a result of testing the dstar lite search algorithm on a path with zero obstacles or walls regardless of size the size of grid selected is less optimal. It only becomes more suitable than the other algorithms when there are changes to the path on the large grid size as a result of my findings.

What this means is that it was the worst algorithm for speed on non-edited paths see (figure 5-7 comparison results) for the findings. Nevertheless even with the time benefits on an edited path on a large grid it cannot be recommended as the most suitable algorithm under a computer games development context as it is considered to be bad games design to have exponential and expansive grids to be searched and it is considered far superior to split up one's search space. This means that it is not likely that a developer would have a grid of this scope and in turn should not use Dstar lite in their game if their grid size is not so expansive.

It should also be noted when the comparisons were made that Dstar Lite and lifelong planning astar were the most difficult algorithms to implement and it should be considered as the need to implement these algorithms into one's game the difficulty of doing so influences my decision in recommending which algorithm and as such cannot recommend the Dstar Lite algorithm under a computer games development context for this reason.

The next thing that has to be taken into account when making my final decision when implementing the pathfinding algorithm is that due to the fact that Dstar Lite is the bigger algorithm with greater depth of code this makes it more difficult to maintain when being compared to that of Astar or Dijkstra's search algorithm of which these algorithms, which do not have the same depth and are easier to maintain and less errors can occur with them.

With all of these factors considered it cannot be recommended that in a game's context dstar lite should be implemented into a game where the speed and memory size of the application is ever paramount and as such it would recommend the depending on the scenario either astar or jump point search depending on the game being developed.

13 Future Work

This section of the paper discusses the potential future work of the project. In this section topics such as the recreation of the project will be discussed and how I would continue my work in the future.

13.1 How the work should be continued?

If I were to go on and continue my research into this topic I would create more data to pull from to fine tune my results in order to solidify my final decision regarding the project topic. I would also add more pathfinding search algorithms into my project to compare against dstar lite. This would include but not limited to Ida star (iterative deepening a star), dstar itself by Anthony Stentz , focused dstar, I would also perhaps include breadth first search and an adapted version of breadth first search to have more non- guided algorithm comparisons against dstar lite. I would also include more visual representation for these algorithms such as the amount of memory allocation each algorithm requires so the user can see this on the screen, and it would help them see more benefits and drawbacks to the use of these algorithms.

13.2 What advice for recreating of my project topic?

However if they are going to start from nothing without the knowledge of this project I would recommend they have a prior understanding of how incremental heuristic-based algorithms work for instance. As this is essential for someone is understanding of how dstar lite works as I did not have any of this information prior to the commencement of the project.

14 References

Swift, N. (2020) *Easy A* (star) pathfinding*, *Medium*. Medium. Available at: https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2 (Accessed: April 24, 2023).

Likhachev, M. *et al.* (2005) *Anytime D* - CMU school of computer science, Carnegie Mellon University School of Computer Science*. DARPA's MARS program, NSF Graduate Research Fellowship. Available at: https://www.cs.cmu.edu/~ggordon/likhachev-etal.anytime-dstar.pdf (Accessed: April 24, 2023).

Stentz , A. (1994) *The D*Algorithm for real-time planning of optimal traverses*. Available at: https://www.ri.cmu.edu/pub_files/pub3/stentz_anthony__tony__1994_2/stentz_anthony__tony__1994_2.pdf (Accessed: April 24, 2023).

Lu, J. *et al.* (2022) *Jump point search algorithm*, *Encyclopedia*. Jiakai Lu. Available at: https://encyclopedia.pub/entry/24246 (Accessed: April 24, 2023).

Raheem, F.A. and Hameed, U.I. (2018) *Path planning algorithm using D* heuristic method based on PSO in ...*, *American Academic Scientific Research Journal for Engineering, Technology, and Sciences*. Available at: https://core.ac.uk/download/pdf/235050716.pdf (Accessed: April 24, 2023).

Harabor, D. and Grastien, A. (2011) *(PDF) improving jump point search - researchgate*. Available at: https://www.researchgate.net/publication/287338108_Improving_jump_point_search (Accessed: April 24, 2023).

Rachmawati, D. and Gustin, L. (2020) *Analysis of Dijkstra's algorithm and a* algorithm in ... - iopscience*. IOP Publishing Ltd. Available at: https://iopscience.iop.org/article/10.1088/1742-6596/1566/1/012061 (Accessed: April 24, 2023).

kaur, N. and Garg, D. (2012) *Analysis of the depth first search algorithms - gdeepak.com*. Available at: https://www.gdeepak.com/pubs/Analysis%20of%20the%20Depth%20First%20Search%20Algorithms.pdf (Accessed: April 24, 2023).

Pathak, M.J., Rami, S.P. and Patel, R.L. (2018) *Comparative analysis of search algorithms - ijcaonline.org*. International Journal of Computer Applications. Available at: https://www.ijcaonline.org/archives/volume179/number50/pathak-2018-ijca-917358.pdf (Accessed: April 24, 2023).

Pandey, K.K. and Kumar, N. (2017) *A comparison and selection on basic type of searching algorithm in data ...*, *researchgate*. Available at: https://www.researchgate.net/publication/308119139_A_Comparison_and_Selection_on_Basic_Type_of_Searching_Algorithm_in_Data_Structure (Accessed: April 24, 2023).

Foead, D. *et al.* (2021) *A systematic literature review of a* pathfinding*, *Procedia Computer Science*. Elsevier. Available at: https://www.sciencedirect.com/science/article/pii/S1877050921000399 (Accessed: April 24, 2023).

Koenig, S., Likhachev, M. and Furcy, D. (2004) *Lifelong planning a∗*, *Artificial Intelligence*. Elsevier. Available at: https://www.sciencedirect.com/science/article/pii/S000437020300225X (Accessed: April 24, 2023).

Koenig, S. and Likhachev, M. (2002) *D* lite - idm-lab.org*. Available at: http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf (Accessed: April 24, 2023).

Patel (2019) *Heuristics from Amit's Thoughts on Pathfinding*, *Heuristics*. Available at: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html (Accessed: April 25, 2023).

15 Appendices

List of data collected.

Description	Images of Data
Astar small Grid Times first 50	(spreadsheet below)

	A
1	0.0000000.135765
2	0.141228
3	0.134918
4	0.136207
5	0.138869
6	0.135377
7	0.137532
8	0.138712
9	0.137757
10	0.13468
11	0.13875
12	0.134387
13	0.139464
14	0.14005
15	0.135846
16	0.13409
17	0.137696
18	0.139591
19	0.143403
20	0.144051
21	0.141763
22	0.142523
23	0.142495
24	0.13392
25	0.136549
26	0.134919
27	0.129693
28	0.13357
29	0.135361
30	0.129606
31	0.133369
32	0.130346
33	0.132238
34	0.134308
35	0.130283
36	0.145112
37	0.1295
38	0.129733
39	0.13471
40	0.136014
41	0.13396
42	0.137701
43	0.133746
44	0.130594
45	0.131788
46	0.129465
47	0.132829
48	0.136345
49	0.132352
50	0.134414

Astar medium Grid Times first 50

1	0.054681
2	0.053883
3	0.054926
4	0.052687
5	0.05154
6	0.053191
7	0.053235
8	0.053147
9	0.052287
10	0.057563
11	0.06216
12	0.053713
13	0.05319
14	0.05194
15	0.051353
16	0.054487
17	0.053711
18	0.052202
19	0.052996
20	0.05813
21	0.053627
22	0.052319
23	0.052203
24	0.053229
25	0.054437
26	0.05482
27	0.05215
28	0.051779
29	0.055283
30	0.053489
31	0.054079
32	0.057263
33	0.051302
34	0.0509
35	0.055435
36	0.051465
37	0.056069
38	0.059032
39	0.053757
40	0.051486
41	0.056076
42	0.053927
43	0.052964
44	0.053137
45	0.052344
46	0.056183
47	0.055833
48	0.051874
49	0.052105
50	0.053751

Astar large Grid Times first 50

1	0.209642
2	0.214689
3	0.211093
4	0.214963
5	0.215844
6	0.211203
7	0.227789
8	0.210902
9	0.2214
10	0.208909
11	0.213074
12	0.208984
13	0.212795
14	0.209617
15	0.215853
16	0.218203
17	0.213802
18	0.210551
19	0.214171
20	0.225907
21	0.21336
22	0.213117
23	0.211189
24	0.211907
25	0.210401
26	0.213306
27	0.209547
28	0.213257
29	0.209192
30	0.213882
31	0.219447
32	0.21237
33	0.207003
34	0.205463
35	0.210572
36	0.212992
37	0.209342
38	0.207072
39	0.217608
40	0.21469
41	0.212128
42	0.214589
43	0.216279
44	0.209876
45	0.217495
46	0.209738
47	0.21266
48	
49	0.189544
50	0.196138

Dstar Lite Small Grid Times First 50

1	0.028686
2	0.034532
3	0.046385
4	0.090104
5	0.035049
6	0.016112
7	0.022683
8	0.030344
9	0.031515
10	0.030867
11	0.02873
12	0.028317
13	0.034318
14	0.033202
15	0.032255
16	0.032101
17	0.029669
18	0.025429
19	0.022392
20	0.14214
21	0.129638
22	0.128363
23	0.142716
24	0.131436
25	0.137311
26	0.12924
27	0.152756
28	0.118592
29	0.11118
30	0.109981
31	0.106102
32	0.114857
33	0.103669
34	0.106228
35	0.108234
36	0.151186
37	0.119174
38	0.113871
39	0.111358
40	0.116575
41	0.119432
42	0.116699
43	0.121257
44	0.138822
45	0.128678
46	0.133479
47	0.114867
48	0.114616
49	0.105249
50	0.106424

Dstar Lite Medium Grid Times First 50

1	0.545594
2	0.607782
3	0.587208
4	0.623153
5	0.614744
6	0.609718
7	0.638352
8	0.604233
9	0.613857
10	0.554224
11	0.534075
12	0.53375
13	0.54831
14	0.546867
15	0.497969
16	0.406852
17	0.443078
18	0.441746
19	0.445767
20	0.4448
21	0.445661
22	0.431747
23	0.446711
24	0.446701
25	0.447706
26	0.447754
27	0.446791
28	0.42962
29	0.431646
30	0.446894
31	0.430666
32	0.4456
33	0.446887
34	0.430237
35	0.447258
36	0.441298
37	1.337874
38	1.333533
39	1.324987
40	1.321093
41	1.319898
42	1.323461
43	1.340371
44	1.320856
45	1.336124
46	1.329389
47	1.328875
48	1.316552
49	1.316467
50	1.32195

Dstar Lite Large Grid Times First 50

1	30.13255
2	15.71816
3	15.67625
4	15.72216
5	0
6	21.50545
7	2.781854
8	2.757896
9	2.726654
10	2.720954
11	2.761993
12	2.208032
13	2.210261
14	2.202767
15	2.181569
16	2.21198
17	2.235227
18	2.209901
19	16.97756
20	17.05391
21	6.50346
22	6.445378
23	22.27615
24	22.22583
25	0.630642
26	0.612634
27	0.60954
28	0.619958
29	0.605822
30	0.628369
31	1.768541
32	1.773555
33	1.757891
34	1.76432
35	1.761563
36	30.28042
37	30.34526
38	12.36236
39	12.38268
40	12.40353
41	12.39844
42	21.40526
43	21.38475
44	1.218921
45	0.341895
46	0.340354
47	0.336397
48	0.338017
49	0.336395
50	0.337124

Dijkstra's Search Small Grid first 50

1	0.004465
2	0.004657
3	0.004215
4	0.004081
5	0.00476
6	0.004344
7	0.004281
8	0.004179
9	0.004119
10	0.004166
11	0.004177
12	0.004104
13	0.004157
14	0.004229
15	0.004345
16	0.0042
17	0.004886
18	0.004064
19	0.004202
20	0.005547
21	0.004153
22	0.004122
23	0.004259
24	0.004135
25	0.004709
26	0.004127
27	0.004131
28	0.004194
29	0.004435
30	0.004595
31	0.004945
32	0.004084
33	0.004128
34	0.004415
35	0.00462
36	0.004248
37	0.005178
38	0.004148
39	0.004063
40	0.004069
41	0.004335
42	0.004417
43	0.004137
44	0.004268
45	0.004379
46	0.004098
47	0.00405
48	0.004123
49	0.004188
50	0.00416

Dijkstra’s Search Medium Grid first 50

1	0.106067
2	0.101982
3	0.100599
4	0.099943
5	0.103269
6	0.104548
7	0.101578
8	0.107218
9	0.103903
10	0.101942
11	0.107162
12	0.107306
13	0.112673
14	0.100669
15	0.107254
16	0.105106
17	0.105172
18	0.110952
19	0.107074
20	0.10653
21	0.109184
22	0.106808
23	0.106779
24	0.106153
25	0.107121
26	0.10463
27	0.118392
28	0.115756
29	0.105299
30	0.103242
31	0.1045
32	0.104414
33	0.110444
34	0.111671
35	0.109614
36	0.102225
37	0.107605
38	0.102765
39	0.105598
40	0.109186
41	0.108401
42	0.113102
43	0.10725
44	0.101297
45	0.103743
46	0.104135
47	0.109744
48	0.103319
49	0.106391
50	0.105074

Dijkstra’s Search Large Grid first 50

1	large0.379594
2	large0.378655
3	large0.381755
4	large0.384210
5	large0.306492
6	large0.314421
7	large0.310135
8	large0.305354
9	large0.310717
10	large0.311183
11	large0.320436
12	large0.301858
13	large0.311105
14	large0.302993
15	large0.310332
16	large0.302764
17	large0.313947
18	large0.303118
19	large0.304529
20	large0.314564
21	large0.306293
22	large0.310978
23	large0.394907
24	large0.415434
25	large0.392459
26	large0.392458
27	large0.408776
28	large0.398304
29	large0.218038
30	large0.210388
31	large0.210789
32	large0.206707
33	large0.205603
34	large0.212545
35	large0.207817
36	large0.207818
37	large0.204675
38	large0.205759
39	large0.211608
40	large0.206845
41	large0.208388
42	large0.211938
43	large0.210890
44	large0.218087
45	large0.209600
46	large0.206759
47	large0.208798
48	large0.213107
49	large0.210722
50	large0.208734

Lifelong Planning Astar Small Grid first 50

1	0.060988
2	0.053446
3	0.052542
4	0.048405
5	0.05359
6	0.056054
7	0.060459
8	0.049954
9	0.016173
10	0.031826
11	0.043666
12	0.019829
13	0.019031
14	0.057758
15	0.019782
16	0.014882
17	0.01341
18	0.019583
19	0.044412
20	0.067002
21	0.058747
22	0.054505
23	0.056759
24	0.014936
25	
26	0.000001
27	0.044518
28	0.064991
29	0.057503
30	0.043987
31	0.052066
32	0.056835
33	0.043385
34	0.03708
35	0.04824
36	0.049523
37	0.047292
38	0.065324
39	0.041056
40	0.05734
41	0.012315
42	0.01315
43	0.052136
44	0.032886
45	0.039039
46	0.047111
47	0.044316
48	0.047052
49	SMALL0.047077
50	SMALL0.012580

Lifelong Planning Astar Medium Grid first 50

1	6.85173
2	0.734882
3	1.834106
4	0.138732
5	0.129428
6	0.124472
7	0.128328
8	0.125412
9	0.128645
10	0.125299
11	0.126269
12	0.141289
13	0.135602
14	0.127624
15	0.133236
16	0.128248
17	0.127459
18	0.127251
19	0.134204
20	0.125799
21	0.13666
22	0.128267
23	0.125909
24	0.122678
25	0.124547
26	0.12436
27	0.127235
28	0.128062
29	0.129842
30	0.126661
31	0.124545
32	0.128332
33	0.126248
34	0.121886
35	0.122602
36	0.121884
37	0.129145
38	0.125402
39	0.12579
40	0.125083
41	0.121427
42	0.122467
43	0.127048
44	0.12334
45	0.123312
46	0.124714
47	0.124113
48	0.127719
49	0.123168
50	0.122851

Lifelong Planning Astar Large Grid first 50

1	31.07115
2	52.64384
3	33.74767
4	40.86055
5	47.72051
6	45.53032
7	53.24764
8	52.95221
9	3.57054
10	4.780712
11	45.7002
12	6.730712
13	0.751454
14	6.681502
15	0.541438
16	0.738923
17	2.816107
18	0.142126
19	0.123806
20	0.127686
21	0.127129
22	0.122066
23	0.129806
24	0.124657
25	0.124153
26	0.125186
27	0.140876
28	0.122897
29	0.125032
30	0.12083
31	0.123917
32	0.121894
33	0.120957
34	0.131863
35	0.127367
36	0.125948
37	0.123203
38	0.122842
39	0.125081
40	0.122601
41	0.1238
42	0.126579
43	0.126039
44	0.125063
45	0.120336
46	0.123575
47	0.124412
48	0.122735
49	0.124408
50	1.169286

Depth First Search Small Grid first 50

	A
1	0.0002
2	1.2824
3	0.9671
4	0.4174
5	0.3823
6	0.2182
7	0.1984
8	0.1839
9	0.1837
10	0.1655
11	0.1837
12	0.2006
13	0.2837
14	0.2487
15	0.1837
16	0.1859
17	0.1802
18	0.3681
19	0.1997
20	0.1998
21	0.1841
22	0.3998
23	0.1993
24	0.3675
25	0.2316
26	0.3511
27	0.199
28	0.1837
29	0.2168
30	2.018
31	0.2998
32	0.0002
33	0.717
34	0.3501
35	0.3163
36	0.251
37	0.216
38	0.4843
39	0.3024
40	0.2962
41	0.2845
42	0.0017
43	0.6325
44	0.1836
45	0.2004
46	0.2178
47	0.1982
48	0.2004
49	0.1998
50	0.2178

Depth First Search Medium Grid first 50

	A
1	0.028
2	0.0548
3	0.0405
4	0.0066
5	0.8858
6	0.1623
7	0.2033
8	0.1751
9	0.2105
10	0.1621
11	0.2193
12	0.1796
13	0.1959
14	0.1734
15	0.1806
16	0.1702
17	0.2087
18	0.2022
19	0.3868
20	0.2042
21	0.2143
22	0.2462
23	0.2321
24	0.2183
25	0.1975
26	0.2247
27	0.2275
28	0.1964
29	0.2224
30	0.2186
31	0.2013
32	0.1922
33	0.2051
34	0.1864
35	0.1995
36	0.1948
37	0.1894
38	0.1923
39	0.1924
40	0.1993
41	0.1951
42	0.2174
43	0.2105
44	0.1902
45	0.1841
46	0.1658
47	0.2024
48	0.1913
49	0.1833
50	0.19

Depth First Search Large Grid first 50

	A
1	0.1844
2	6.4758
3	0.6369
4	0.364
5	0.2801
6	0.2544
7	0.2033
8	0.3693
9	0.2056
10	0.3639
11	0.161
12	0.3703
13	0.2023
14	0.2413
15	0.1671
16	0.2047
17	0.3656
18	0.2105
19	0.1632
20	0.4405
21	0.1999
22	0.1664
23	0.5239
24	0.376
25	0.2293
26	0.2012
27	0.1633
28	0.201
29	0.2019
30	0.1985
31	0.2486
32	0.2426
33	0.2025
34	0.201
35	0.1652
36	0.4071
37	0.1648
38	0.2082
39	0.2074
40	0.2105
41	0.2108
42	0.6469
43	0.2846
44	0.2537
45	0.2423
46	0.2429
47	0.2056
48	0.2133
49	0.2443
50	0.2485