Computer Games Development
Project Report
Year IV

Donal Howe
C00249662
[Date of Submission]
[Declaration form to be attached]

Table Of Contents

List Of Figures

List Of Tables

Acknowledgements

I would like to thank my project supervisor Oisin Crawley 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 more enjoyable.

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 conclude whether the algorithms chosen when put against the Dstar Lite algorithm are a better alternative to the dynamic pathfinding algorithm or perhaps it may be the case where Dstar Lite is the more applicable algorithm for the scenario presented in the paper.

Not only will the paper provide the times in which it took these algorithms to traverse these scenarios to find the shortest path it will also talk about the implementation itself and if the degree of difficulty which was involved in the implementation of each algorithm influences the decision of the user to implement the 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.

1 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 and 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 onto their game world in order to try and find which one suited their game and its environment the best which could potentially be quite time inducive and what my research project hopes to achieve is to compare several of these different pathfinding algorithms on 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 by this it is not meant whether the grid is one dimensional or two dimensional by this statement it means what size of grid they have chosen to implement, whether the world is changing dynamically and how this will affect the grid which in turn would affect the times it takes for these algorithms to complete and their memory usage will increase and how they handle these changes. This may for instance destroy a Cell on the grid by placing a wall or some sort of obstacle 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 several heuristic and non-heuristic algorithms against Dstar Lite on a dynamic grid and how the data which has been collected because of the research may have an impact on developers and the games that they are or could developing due to the algorithms that they implement perhaps shine in different scenarios to others. It will do this as in my project it does allow a visual comparison between algorithms what is mean by this is that the user will be able to see the algorithm that they choose race against Dstar Lite in real time and the path which they take. 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.

My end goal of this research topic is to comprehensively and conclusively come to the most optimal decision for the reader of this project and then in turn they will be able to go and implement the algorithm most suited to their problem which will 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.

2 Research Question

The main purpose of this research is to find out which algorithm is best suited for traversing a dynamic 2D grid in a game world and find out their benefits and drawbacks. The algorithms which have been investigated for this project are the following “Dstar Lite” search algorithm, “Lifelong planning Astar,” “Jump Point Search,” “Dijkstra’s search algorithm,” and “Depth first Search” pathfinding algorithm.

3 Objectives involved in the making of this project.

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

3.1 Technologies to be used:

SFML -2.5.1

Visual Studio 2022

C++

3.2 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

4 Literature Review

4.1 Background.

This section of the paper presents a review on the pathfinding algorithms which have been implemented into the application. These concepts and understandings were needed to make a comprehensive and complete comparison of these algorithms on a 2D game world grid. These understandings allowed me to form a visual representation of the algorithms in the project.

4.2 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 on 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 shortest path along the grid world for the character to traverse. Which in turn makes the character movement more efficient than just manually moving them once they either hit into a wall or change their direction accordingly to 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?

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

directed pathfinding algorithm does however spend resources on trying to figure out both how far it is away from the destination and the start. What they do is look around and access all the neighbouring nodes edge costs and 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 feature which have been used as a part of my implementation. More key features which are needed for the readers understanding will be explained in more detail which is to follow.

4.3 Dstar Lite Search Pathfinding Algorithm

4.3.1 Overview

Dstar Lite is an incremental heuristic pathfinding search algorithm which allows for the replanning of the 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.

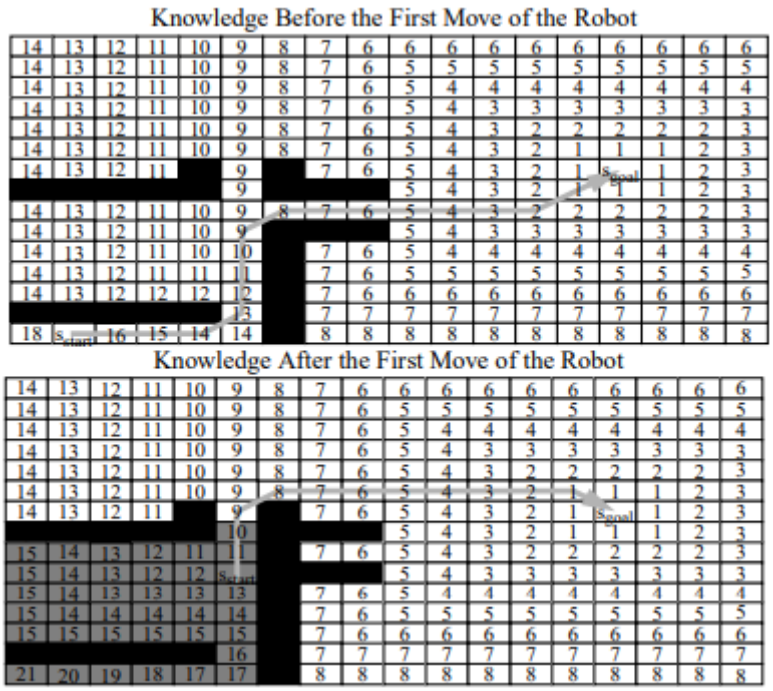

Figure 4-1 Dstar lite path.

4.3.2 Key Information

Key variables which need to be understood for the Dstar Lite algorithm. One thing to understand is that each node in the grid has these variables contained within them so 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 is on the given graph/grid to the start node.

“Hcost” – the Hcost of a Node is the distance from where that node is on the graph/grid to the destination/goal node.

“Rhs cost” – rhs cost otherwise known as the right-hand side value is used with a different understanding in dstar lite to 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”- key modifier found in the application as “K_M” 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.

4.3.3 Algorithm

```
procedure CalculateKey(s)
{01'} return [min(g(s), rhs(s)) + h(s_start, s) + k_m ; min(g(s), rhs(s))];
procedure Initialize()
{02'} U = ∅;
{03'} k_m = 0;
{04'} for all s ∈ S rhs(s) = g(s) = ∞;
{05'} rhs(s_goal) = 0;
{06'} U.Insert(s_goal, CalculateKey(s_goal));
procedure UpdateVertex(u)
{07'} if (u ≠ s_goal) rhs(u) = min_{s' ∈ Succ(u)}(c(u, s') + g(s'));
{08'} if (u ∈ U) U.Remove(u);
{09'} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
{10'} while (U.TopKey() < CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start))
{11'}   k_old = U.TopKey();
{12'}   u = U.Pop();
{13'}   if (k_old < CalculateKey(u))
{14'}     U.Insert(u, CalculateKey(u));
{15'}   else if (g(u) > rhs(u))
{16'}     g(u) = rhs(u);
{17'}     for all s ∈ Pred(u) UpdateVertex(s);
{18'}   else
{19'}     g(u) = ∞;
{20'}     for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);
procedure Main()
{21'} s_last = s_start;
{22'} Initialize();
{23'} ComputeShortestPath();
{24'} while (s_start ≠ s_goal)
{25'}   /* if (g(s_start) = ∞) then there is no known path */
{26'}   s_start = arg min_{s' ∈ Succ(s_start)}(c(s_start, s') + g(s'));
{27'}   Move to s_start;
{28'}   Scan graph for changed edge costs;
{29'}   if any edge costs changed
{30'}     k_m = k_m + h(s_last, s_start);
{31'}     s_last = s_start;
{32'}     for all directed edges (u, v) with changed edge costs
{33'}       Update the edge cost c(u, v);
{34'}       UpdateVertex(u);
{35'}     ComputeShortestPath();
```

Figure 4-2 Dstar lite algorithm.

[1]How does the algorithm work? Dstar Lite is an extension of Lifelong planning astar and is an incremental algorithm. Dstar rather than a typical search algorithm which searches from the star to the destination node, Dstar does not do this it searches backwards from the destination node to the goal node. Once a best path is found from destination to start the start position or where the robot is currently is 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 heuristics 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 otherwise after the robot has moved if our key modifier were not implemented into us calculate key function the nodes heuristic value would not be correct once a recalculation had to be made.

[1] Koenig, S. and Likhachev, M. (no date) *D* lite - idm-lab.org*, *D* Lite*. Available at: http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf (Accessed: April 11, 2023).

How is the shortest path found? we are dealing with two separate states of a node which are consistent and inconsistent nodes. What this means is a node is consistent it means that there are no current changes need to be made to it as its Gcost value and Rhs cost are equal in turn its values are consistent with one another.

Then there is inconsistent which means that a nodes G cost values and Rhs values are not equal and as such we must deal with this to progress the search. Within this bracket of states there is two distinct types of inconsistencies. There is over consistency where our Gcost is Greater than our Rhs cost and there is under consistent where our Rhs values is greater than our Gcost value.

How do we deal with this? First thing is to 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 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 readd the node to back into our priority queue.

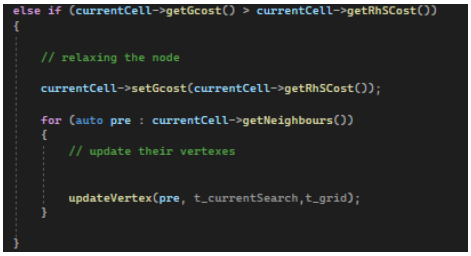

Figure 4-3 Dealing with over consistencies.

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.

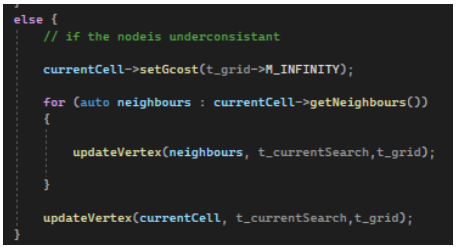

Figure 4-4 Dealing with under consistencies.

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.

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.

```
if (a->m_key.first > b->m_key.first) {
    return true;
}
```

Figure 4-5 Function comparison.

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.

```
else if (a->m_key.first == b->m_key.first && a->m_key.second > b->m_key.second) {
    return true;
}
```

Figure 4-6 Dealing with ties.

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 have increased edge costs and are as such now not able to be traversed, we will recompute a best search around the node.

```
// checking for any edge cost changes of the surrounding neighbours

for (auto neighbours : t_start->getNeighbours())
{
    if (neighbours->getTraversable() == false)
    {
        K_M = K_M + t_grid->heuristic(s_Last, t_start);
        s_Last = t_start;
        updateVertex(neighbours, t_currentSearch, t_grid);

    }
    ComputeShortestPath(t_start, t_currentSearch, t_grid);
}
```

Figure 4-7 Dealing with introduction of untraversable.

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.

```
std::pair<double, double> DstarLite::calculateDstarKey(Cell* t_currentSearch, Cell* Start,Grid * t_grid)
{

    double heuristicVal = t_grid->heuristic(t_currentSearch, Start) + K_M;
    double minVal = std::min(t_currentSearch->getGcost(), t_currentSearch->getRhSCost());
    std::pair<double, double> temp1 = std::make_pair(heuristicVal + minVal, std::min(t_currentSearch->getGcost(), t_currentSearch->getRhSCost()));
    return temp1;
}
```

Figure 4-8 Calculating a node key.

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 Dstar behaves 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:

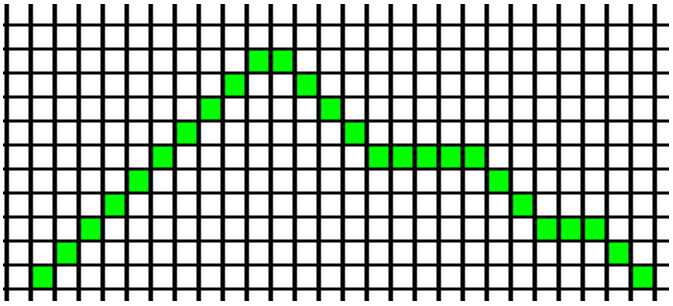

Figure 4-9 Dstar lite path on 2D grid.

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 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.

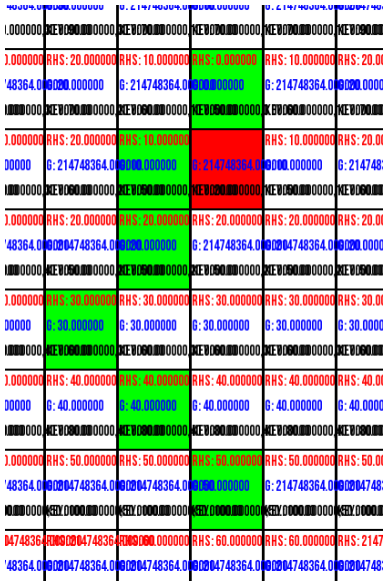

Figure 4-10 How Dstar path changes with wall on path.

That is an overview of the Dstar Lite algorithm and how it works on a 2D dynamic grid aswell as how it behaves when a wall is placed on the given path

4.4 Astar Search Pathfinding Algorithm.

4.4.1 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.

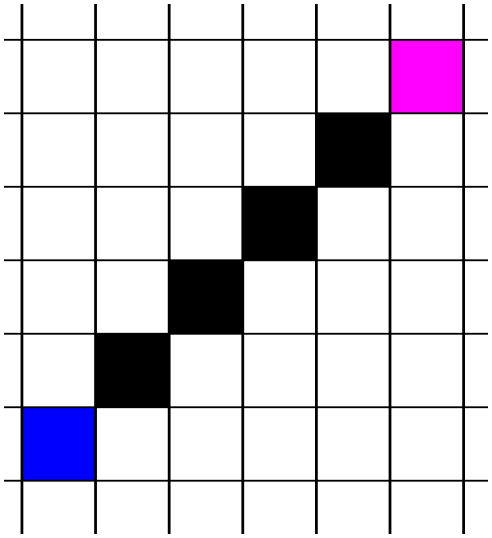

Figure 4-11Astar path.

4.4.2 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 (mentioned in 6.3.2) plus the value of the Hcost (mentioned in 6.3.2). what this value does is allow us to easily investigate a certain node based on its location in the grid.

“Gcost” – (mentioned in 6.3.2)

“Hcost” – (mentioned in 6.3.2)

“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 “If h(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.

4.4.3 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.

```
bool operator()(Cell* t_n1, Cell* t_n2) const
{
    return (t_n1->getGcost() + t_n1->getHcost()) > (t_n2->getGcost() + t_n2->getHcost());
}
```

Figure 4-12 Fcost function.

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 figure below(figure 6-3 Astar algorithm)

```
while (pq.size() != 0 && pq.top() != goal)
{
    Cell* topnode = pq.top();

    for (Cell* q : topnode->getNeighbours())
    {

        Cell* child = q;

        if (child != pq.top()->GetPrev())
        {

            int weight = child->getWeight();

            int distanceToChild = pq.top()->getGcost() + weight;

            if (distanceToChild < child->getGcost() && child->getTraversable() == true)
            {
                child->setGcost(distanceToChild);
                child->setPrev(pq.top());
                if (child == goal)
                {
                    child->setColor(sf::Color::Magenta);
                    AstarDone = true;
                    m_Astartimer = m_clock.getElapsedTime();
                }
            }
            if (child->getMarked() == false)
            {
                pq.push(child);

                child->setMarked(true);
            }

        }
    }

    pq.pop();
}
```

Figure 4-13 Astar algorithm.

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

4.5 Lifelong planning Astar Search Pathfinding Algorithm.

4.5.1 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

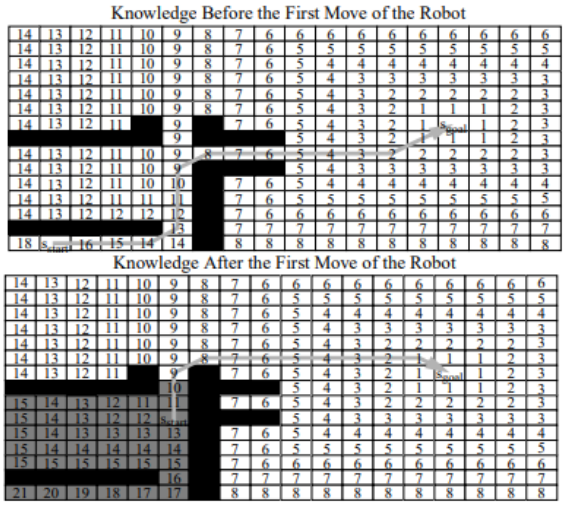

Figure 4-14 Lpa* path.

4.5.2 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” – as mentioned in (6.3.2)

“Gcost” – as mentioned in (6.3.2)

“Key Value” – as mentioned in (6.3.2), 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.

4.5.3 Algorithm

procedure CalculateKey(s)
{01} return $[\min(g(s), rhs(s)) + h(s, s_{goal}); \min(g(s), rhs(s))]$;
procedure Initialize()
{02} $U = \emptyset$;
{03} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{04} $rhs(s_{start}) = 0$;
{05} U.Insert(s_{start}, CalculateKey(s_{start}));
procedure UpdateVertex(u)
{06} if $(u \neq s_{start})$ $rhs(u) = \min_{s' \in Pred(u)}(g(s') + c(s', u))$;
{07} if $(u \in U)$ U.Remove(u);
{08} if $(g(u) \neq rhs(u))$ U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
{09} while (U.TopKey() $<$ CalculateKey(s_{goal}) OR $rhs(s_{goal}) \neq g(s_{goal})$)
{10} u = U.Pop();
{11} if $(g(u) > rhs(u))$
{12} $g(u) = rhs(u)$;
{13} for all $s \in Succ(u)$ UpdateVertex(s);
{14} else
{15} $g(u) = \infty$;
{16} for all $s \in Succ(u) \cup \{u\}$ UpdateVertex(s);
procedure Main()
{17} Initialize();
{18} forever
{19} ComputeShortestPath();
{20} Wait for changes in edge costs;
{21} for all directed edges (u, v) with changed edge costs
{22} Update the edge cost $c(u, v)$;
{23} UpdateVertex(v);

Figure 4-15 Lpa* algorithm

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.

```
std::pair<double, double> LpaStar::calculateKey(Cell* s, Cell* t_goal,Grid *t_grid)
{

    double g_rhs = std::min(s->getGcost(), s->getRhSCost());
    double h = t_grid->heuristic(s, t_goal);

    return std::make_pair(g_rhs + h , g_rhs);
}
```

Figure 4-16 The calculation of the key.

Figure 7-3 The calculation of the key

While the open list which has been populated with the start node is note empty.

[2] Koenig, S. and Likhachev, M. (no date) *D* lite - idm-lab.org*, *D* Lite*. Available at: http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf (Accessed: April 11, 2023).

Remove the node with the smallest key values which is compared via the functor as such. The functor corrects tie values in the priority queue by returning the node highest in the queue already.

```
class KeyComparer {
public:
    bool operator()(const Cell* a, const Cell* b) const {
        if (a->m_key.first < b->m_key.first) {
            return true;
        }
        else if (a->m_key.first == b->m_key.first && a->m_key.second < b->m_key.second) {
            return true;
        }
        else {
            return false;
        }
    }
};
```

Figure 4-17 Lpa* functor.

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.

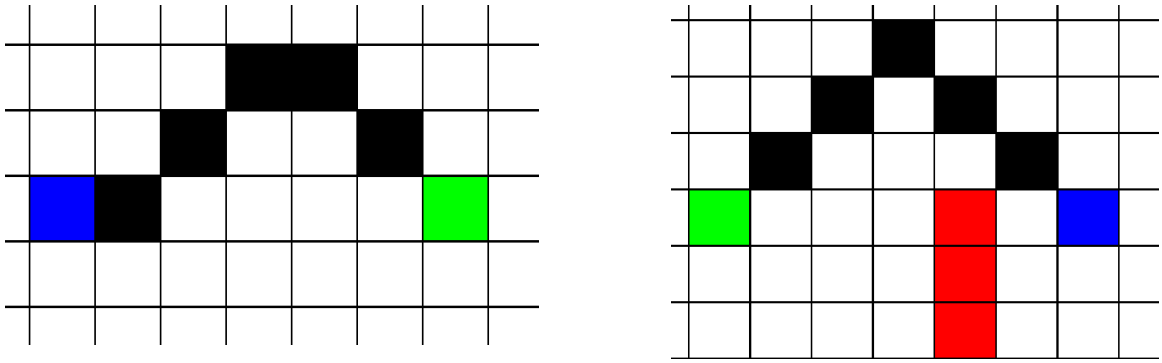

Figure 4-18 Lpa* path without walls vs with walls.

4.6 Dijkstra's Search Pathfinding Algorithm

4.6.1 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.

4.6.2 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 (6.3.2)

“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 (6.5.2)

“Previous pointer” – as mentioned in (6.4.2)

“weight” – as mentioned in (6.4.2)

4.6.3 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 (seen in figure 8-1).

```
bool operator()(Cell* t_n1, Cell* t_n2) const
{
    return (t_n1->getGcost()) > (t_n2->getGcost());
}
```

Figure 4-19 Dijkstra's functor.

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.

```
s->setGcost(0);
s->setStartColour();
pq.push(s);
pq.top()->setMarked(true);

while (pq.size() != 0 && pq.top() != g)
{

    auto iter = pq.top()->getNeighbours().begin();
    auto endIter = pq.top()->getNeighbours().end();

    for (; iter != endIter; iter++)
    {

        Cell* child = (*iter);
        if (child != pq.top()->GetPrev())
        {

            int distanceToChild = ((*iter)->getWeight() + pq.top()->getGcost());

            if (distanceToChild < child->getGcost() && child->getTraversable() == true)
            {
                child->setGcost(distanceToChild);
                child->setPrev(pq.top());
                if (child == t_Goal) {
                    std::cout << "djikstras" << std::endl;
                    DjkstrasTimer = m_clock.getElapsedTime();
                    djkstrasPathFound = true;
                }

            }
            if (child->getMarked() == false)
            {
                pq.push(child);

                child->setMarked(true);
            }
        }
    }

    pq.pop();
}
```

Figure 4-20 Dijkstra's search algorithm.

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 8-3 Dijkstra's path without wall) and (Figure 8-4 Dijkstra's path with wall on path)

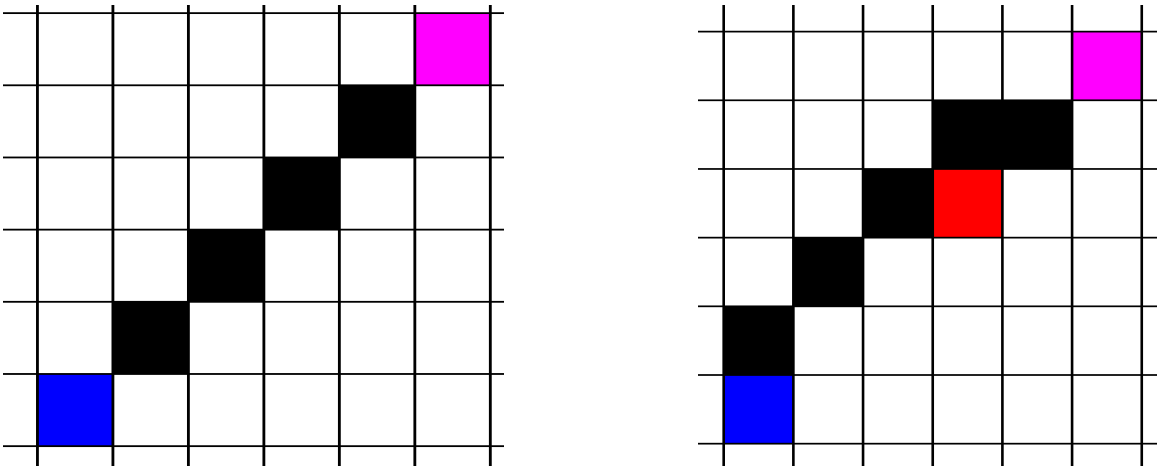

Figure 4-21 Dijkstra's path without walls vs with walls.

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.

4.7 Depth first Search Pathfinding Algorithm

4.7.1 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.

4.7.2 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 (6.4.2) this is used for tracking the path taken.

4.7.3 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.

```
if (nullptr != t_curr && depthGoalFound == false) {
    // process the current node and mark it
    std::cout << t_curr->getID() << std::endl;
    t_curr->setMarked(true);


    for (auto itr = t_curr->getNeighbours().begin(); itr != t_curr->getNeighbours().end(); itr++)
    {
        if ((*itr)->getTraversable() == false)
        {
            continue;
        }
        // process the linked node if it isn't already marked.
        if ((*itr) == t_goal)
        {
            (*itr)->setPrev(t_curr);
            std::cout << "found goal" << std::endl;
            depthfirstSearchTimer = m_clock.getElapsedTime();
            depthGoalFound = true;
            break;
        }
        if ((*itr)->getMarked() == false)
        {
            (*itr)->setPrev(t_curr);
            computeShortestPath((*itr), t_goal,t_grid);

        }
    }
}
```

Figure 4-22 Depth first search algorithm.

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.

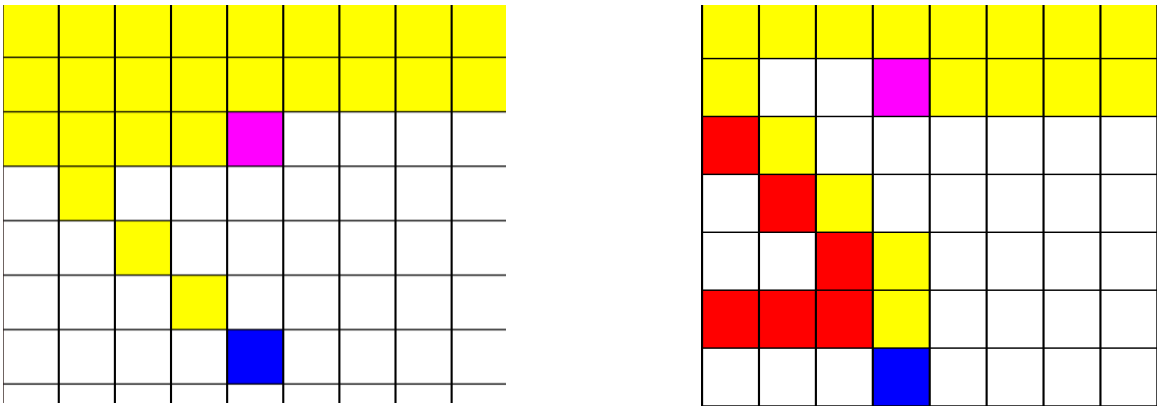

Figure 4-23 Depth first search algorithm without walls vs with walls.

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.

4.8 Jump point search Pathfinding Algorithm

4.8.1 Overview

4.8.2 Key Information

4.8.3 Algorithm

5 Study/Methodology

This section of the paper shows how theories of pathfinding algorithms researched in section 4 were implanted in this study.

5.1 Implementing Astar

5.2 Implementing Dstar

6 Evaluation and Discussion

This section of the paper demonstrates the results gather from the trials performed to develop the ability compare the algorithms used in the project. It will show these results and discuss the factors which led to them and how it influences the decisions that could be made when making a decision on whether to implement each algorithm.

6.1 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.

6.2 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.

6.3 How will the data be displayed?

When comparing the times in which it took for each search algorithm to find the destination/ goal node, 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 seperate times for each sized grid 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 bias potentially given favor 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 scenario for the algorithms was collected and calculated. For collecting the data 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.

On a small grid the first 50 times for astar.

7 Results

7.1 Calculation of average times examples

7.1.1 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

Table 7-1 Small grid times 0 walls

Astar search algorithm average time.

0.000961 + 0.001055 + 0.001033 + 0.000978 + 0.000968 + 0.00095 + 0.00097 + 0.000997 + 0.000964 + 0.000953 + 0.000996 + 0.000825 + 0.000842 + 0.001239 + 0.000975 + 0.000906 + 0.000826 + 0.001081 + 0.000831 + 0.000839 + 0.000858 + 0.000829 + 0.000845 + 0.000838 + 0.00089 + 0.00085 + 0.000856 + 0.000833 + 0.000837 + 0.00083 + 0.000877 + 0.001162 + 0.000843 + 0.000878 + 0.000884 + 0.000835 + 0.00086 + 0.000804 + 0.001674 + 0.00084 + 0.000847 + 0.000809 + 0.000827 + 0.00087 + 0.000883 + 0.000849 + 0.001244 + 0.001234 + 0.000862 + 0.000844 + 0.000812 + 0.000851

= 0.045156

0.045156 / 50

Average = 0.00090312

Dstar lite search algorithm average time.

0.028686 + 0.034532 + 0.046385 + 0.090104 + 0.035049 + 0.016112 + 0.022683 + 0.030344 + 0.031515 + 0.030867 + 0.02873 + 0.028317 + 0.034318 + 0.033202 + 0.032255 + 0.032101 + 0.029669 + 0.025429 + 0.022392 + 0.14214 + 0.129638 + 0.128363 + 0.142716 + 0.131436 + 0.137311 + 0.12924 + 0.152756 + 0.118592 + 0.11118 + 0.109981 + 0.106102 + 0.114857

+ 0.103669 + 0.106228 + 0.108234 + 0.151186 + 0.119174 + 0.113871 + 0.111358 + 0.116575 + 0.119432 + 0.116699 + 0.121257 + 0.138822 + 0.128678 + 0.133479 + 0.114867 + 0.114616 + 0.105249 = 2.56238

2.56238 / 47

Average = 0.06404

Dijkstra's Search algorithm average time.

(0.004465 + 0.004657 + 0.004215 + 0.004081 + 0.00476 + 0.004344 + 0.004281 + 0.004179 + 0.004119 + 0.004166 + 0.004177 + 0.004104 + 0.004157 + 0.004229 + 0.004345 + 0.0042 + 0.004886 + 0.004064 + 0.004202 + 0.005547 + 0.004153 + 0.004122 + 0.004259 + 0.004135 + 0.004709 + 0.004127 + 0.004131 + 0.004194 + 0.004435 + 0.004595 + 0.004945 + 0.004084 + 0.004128 + 0.004415 + 0.00462 + 0.004248 + 0.005178 + 0.004148 + 0.004063 + 0.004069 + 0.004335 + 0.004417 + 0.004137 + 0.004268 + 0.004379 + 0.004098 + 0.00405 + 0.004123 + 0.004188 + 0.00416) / 50

Average = 0.0043366

Lifelong planning astar average time.

0.060988 + 0.053446 + 0.052542 + 0.048405 + 0.05359 + 0.056054 + 0.060459 + 0.049954 + 0.016173 + 0.031826 + 0.043666 + 0.019829 + 0.019031 + 0.057758 + 0.019782 + 0.014882 + 0.01341 + 0.019583 + 0.044412 + 0.067002 + 0.058747 + 0.054505 + 0.056759 + 0.014936 + 0.000001 + 0.044518 + 0.064991 + 0.057503 + 0.043987 + 0.052066 + 0.056835 + 0.043385 + 0.03708 + 0.04824 + 0.049523 + 0.047292 + 0.065324 + 0.041056 + 0.05734 + 0.012315 + 0.01315 + 0.052136 + 0.032886 + 0.039039 + 0.047111 + 0.044316 + 0.044052 + 0.016331 + 0.051738 + 0.012982 + 0.010882 = 2.317497

2.317497 / 50

Average = 0.0463567435897436

7.1.2 Medium grid times with 0 Walls

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

Table 7-2 Medium grid times 0 walls

Table 7-3 Medium grid times 0 walls

Astar search algorithm average time.

(0.054681 + 0.053883 + 0.054926 + 0.052687 + 0.05154 + 0.053191 + 0.053235 + 0.053147 + 0.052287 + 0.057563 + 0.06216 + 0.053713 + 0.05319 + 0.05194 + 0.051353 + 0.054487 + 0.053711 + 0.052202 + 0.052996 + 0.05813 + 0.053627 + 0.052319 + 0.052203 + 0.053229 + 0.054437 + 0.05482 + 0.05215 + 0.051779 + 0.055283 + 0.053489 + 0.054079 + 0.057263 + 0.051302 + 0.0509 + 0.055435 + 0.051465 + 0.056069 + 0.059032 + 0.053757 + 0.051486 + 0.056076 + 0.053927 + 0.052964 + 0.053137 + 0.052344 + 0.056183 + 0.055833 + 0.051874 + 0.052105 + 0.053751) / 50

Average = 0.053890

Dstar lite search algorithm average time.

0.545594 + 0.607782 + 0.587208 + 0.623153 + 0.614744 + 0.609718 + 0.638352 + 0.604233 + 0.613857 + 0.554224 + 0.534075 + 0.53375 + 0.54831 + 0.546867 + 0.497969 + 0.406852 + 0.443078 + 0.441746 + 0.445767 + 0.4448 + 0.445661 + 0.431747 + 0.446711 + 0.446701 + 0.447706 + 0.447754 + 0.446791 + 0.42962 + 0.431646 + 0.446894 + 0.430666 + 0.4456 + 0.446887 + 0.430237 + 0.447258 + 0.441298 + 1.337874 + 1.333533 + 1.324987 + 1.321093 + 1.319898 + 1.323461 + 1.340371 + 1.320856 + 1.336124 + 1.329389 + 1.328875 + 1.316552 + 1.316467 + 1.32195

= 19.738098

19.738098 / 41

Average =0.4816679024

Dijkstra's Search algorithm average time.

(0.060988 + 0.053446 + 0.052542 + 0.048405 + 0.05359 + 0.056054 + 0.060459 + 0.049954 + 0.016173 + 0.031826 + 0.043666 + 0.019829 + 0.019031 + 0.057758 + 0.019782 + 0.014882 + 0.01341 + 0.019583 + 0.044412 + 0.067002 + 0.058747 + 0.054505 + 0.056759 + 0.014936 + 0.000001 + 0.044518 + 0.064991 + 0.057503 + 0.043987 + 0.052066 + 0.056835 + 0.043385 + 0.03708 + 0.04824 + 0.049523 + 0.047292 + 0.065324 + 0.041056 + 0.05734 + 0.012315 + 0.01315 + 0.052136 + 0.032886 + 0.039039 + 0.047111 + 0.044316 + 0.047052 + 0.016331 + 0.051738 + 0.012982 + 0.010882) / 50

Average = 0.03953446

Lifelong planning astar search algorithm average time.

(1.834106 + 0.138732 + 0.129428 + 0.124472 + 0.128328 + 0.125412 + 0.128645 + 0.125299 + 0.126269 + 0.141289 + 0.135602 + 0.127624 + 0.133236 + 0.128248 + 0.127459 + 0.127251 + 0.134204 + 0.125799 + 0.13666 + 0.128267 + 0.125909 + 0.122678 + 0.124547 + 0.12436 + 0.127235 + 0.128062 + 0.129842 + 0.126661 + 0.124545 + 0.128332 + 0.126248 + 0.121886 + 0.122602 + 0.121884 + 0.129145 + 0.125402 + 0.12579 + 0.125083 + 0.121427 + 0.122467

+ 0.127048 + 0.12334 + 0.123312 + 0.124714 + 0.124113 + 0.127719 + 0.123168 + 0.122851 + 0.122949 + 0.125599) =3.168434

3.168434 / 50

Average = 0.06336868

7.1.3 Large grid times with 0 Walls

Algorithm	Average Time (Seconds)	Ranking Based on Quickest Average Time (Seconds)
Astar	0.18156930	1
Dstar	4.03639998	4
Dijkstra's	0.34433232	2
Lifelong planning Astar	1.3943158823529413	3

Table 7-4 Large grid times 0 walls

Astar search algorithm average time.

0.209642 + 0.214689 + 0.211093 + 0.214963 + 0.215844 + 0.211203 + 0.227789 + 0.210902 + 0.2214 + 0.208909 + 0.213074 + 0.208984 + 0.212795 + 0.209617 + 0.215853 + 0.218203 + 0.213802 + 0.210551 + 0.214171 + 0.225907 + 0.21336 + 0.213117 + 0.211189 + 0.211907 + 0.210401 + 0.213306 + 0.209547 + 0.213257 + 0.209192 + 0.213882 + 0.219447 + 0.21237 + 0.207003 + 0.205463 + 0.210572 + 0.212992 + 0.209342 + 0.207072 + 0.217608 + 0.21469 + 0.212128 + 0.214589 + 0.216279 + 0.209876 + 0.217495 + 0.209738 + 0.21266 + 0.190009 + 0.189544 + 0.196138 = 8.525751

8.525751 / 47

Average =0.18156930

Dstar lite search algorithm average time.

(2.781854 + 2.757896 + 2.726654 + 2.720954 + 2.761993 + 2.208032 + 2.210261 + 2.202767 + 2.181569 + 2.21198 + 2.235227 + 2.209901 + 16.97756 + 17.053909 + 6.50346 + 6.445378 + 22.27615 + 22.225832 + 0.630642 + 0.612634 + 0.60954 + 0.619958 + 0.605822 + 0.628369 + 1.768541 + 1.773555 + 1.757891 + 1.76432 + 1.761563 + 30.28042 + 30.345257 + 12.362361 + 12.382677 + 12.40353 + 12.398441 + 21.405256 + 21.384754 + 1.218921 + 0.341895 + 0.340354 + 0.336397 + 0.338017 + 0.336395 + 0.337124 + 0.347263 + 0.335846 + 0.345248 + 0.349882 + 0.341001 + 0.339432 + 0.345655 + 0.339069 + 0.336853 + 0.338817 + 0.341829) / 50 = 201.819999

201.819999 / 50

Average =4.03639998

Dijkstra's Search algorithm average time.

0.379694 + 0.378655 + 0.381755 + 0.38421 + 0.306492 + 0.314421 + 0.310135 + 0.305354 + 0.310717 + 0.311183 + 0.320436 + 0.301858 + 0.311105 + 0.302993 + 0.310332 + 0.302764 + 0.313947 + 0.303118 + 0.304529 + 0.314564 + 0.306293 + 0.310978 + 0.394907 + 0.415434 + 0.392459 + 0.392458 + 0.408776 + 0.398304 + 0.218038 + 0.210388 + 0.210789 + 0.206707 + 0.205603 + 0.212545 + 0.207817 + 0.207818 + 0.204675 + 0.205759 + 0.211608 + 0.206845 + 0.208388 + 0.211938 + 0.21089 + 0.218087 + 0.2096 + 0.206759 + 0.208798 + 0.213107 + 0.210722 + 0.208734 = 17.216616

17.216616 / 50

Average = 0.34433232

Lifelong planning astar search algorithm average time.

4.780712 + 45.700203 + 6.730712 + 0.751454 + 6.681502 + 0.541438 + 0.738923 + 2.816107 + 0.142126 + 0.123806 + 0.127686 + 0.127129 + 0.122066 + 0.129806 + 0.124657 + 0.124153 + 0.125186 + 0.140876 + 0.122897 + 0.125032 + 0.12083 + 0.123917 + 0.121894 + 0.120957 + 0.131863 + 0.127367 + 0.125948 + 0.123203 + 0.122842 + 0.125081 + 0.122601 + 0.1238 + 0.126579 + 0.126039 + 0.125063 + 0.120336 + 0.123575 + 0.124412 + 0.122735 + 0.124408 + 1.169286 + 3.256999 + 0.133915 + 0.127328 + 0.129216 + 0.122407 + 0.123211 + 0.132402 + 0.121082 + 0.123017 + 0.134808 = 71.110201

Average = 1.3943158823529413

Those are the average values generated on running these algorithms at random positions with no walls present on the path so there were no recalculations required by any of the algorithms.

Small grid size best average time- astar search algorithm with 0.00090312

Small grid size worst average time – Dstar Lite search algorithm with 0.06404

Medium gird size best average time- Dijkstra's search algorithm with 0.03953446

Medium grid size worst average time – Dstar Lite search algorithm with 0.4816679024

Large grid size best average time- Astar search algorithm with 0.18156930

Large gird size worst average time - Dstar Lite search algorithm with 4.03639998

///// need to add times with walls as this is v important for dstar

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 7-5 Comparison results.

The next from of comparison that is to be discussed is about is the difficulty of implementation. This form of comparison influenced my findings as in games development management of resources is paramount and the difficulty of implementing these algorithms can influence a project leads decision when implementing these algorithms. Through this form of comparison the topic of an algorithm's robustness will be covered and the complexity of them or the depth of them.

8 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.

9 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)

10 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.

11 Conclusions and discussion of results

To conclude this report this chapter will cover the decision and the corresponding reasoning behind it. To answer the question on whether it would be optimal to use dstar lite as the pathfinding algorithm of choice in a game.

As the result of the data collected and compared it is not believed that it would be and here is why. One from the data which has been put forward dstar lite on a path with zero obstacles or walls regardless of size is less optimal and only comes into its own once a correction needs to be made to the path however as found above the time difference on the small and medium sized grids is not of such a magnitude to where it is warranted to be selected for use. Even though its benefits on the larger grid with more walls on it is more substantial this is not very realistic in a game's development context let me explain. In a game regardless of genre you are not highly likely to have a grid of such magnitude and if you were to have one it is believed to be bad overall design as you should split up your search space to where you will not have a grid of such size. So with that considered the benefits seen to dstar lite are not as likely to be seen as a result in a game. There is also the factor which needs to be considered that dstar lite is the harder algorithm to be implemented into the game and could potentially lead to more errors when the developer needs to try and maintain the code.

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.

12 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.

12.1 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.

12.2 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.

13 References

https://core.ac.uk/download/pdf/235050716.pdf - Path Planning Algorithm using D* Heuristic Method Based on PSO in Dynamic Environment Firas A. Raheema *, Umniah I. Hameedb

https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2 -Nicholas Swift Feb 27 2017

http://www.cs.cmu.edu/~ggordon/likhachev-etal.anytime-dstar.pdf - Maxim Likhachev† , Dave Ferguson† , Geoff Gordon† , Anthony Stentz† , and Sebastian Thrun‡

https://www.ri.cmu.edu/pub_files/pub3/stentz_anthony__tony__1994_2/stentz_anthony__tony__1994_2.pdf -Anthony Stentz

Koenig, S. and Likhachev, M. (n.d.). *D* Lite*. [online] Available at: http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf

encyclopedia.pub. (n.d.). *Jump Point Search Algorithm*. [online] Available at: https://encyclopedia.pub/entry/24246

Patel, A. (2019). *Heuristics*. [online] Stanford.edu. Available at: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

14 14.0 Appendices

Replace this text with Appendices.

This might include ethics application and other relevant material e.g. copy of any questionnaires used.