Computer Games Development SE607
Software Functional Specification
Year IV

Donal Howe [Student Name]
C00249662 [Student Number]

[Date of Submission]

[Declaration form to be attached]

Table Of Contents

1.0 Introduction

The objective of this project was to put together a comprehensive comparison of guided and non-guided based pathfinding algorithms to the incremental dynamic pathfinding algorithm known as Dstar Lite under a game's development context. So one can decide based of the information shown in this document whether to or not implement Dstar lite into their project or perhaps to implement another algorithm such as life long planning A star or, A star itself.

1.1 Brief description of the chosen algorithms

1.1.1 Description of D star Lite

Dstar Lite works as a dynamic A star where it can make changes to the path along the graph without having to rerun the process of calculating the path. Where Astar has to calculate the heuristic (cost of the node form the destination + the cost of the node from the beginning node) for each node upon running the algorithm to find the shortest path D star does not. It works by only investigating nodes which have been affected by a non-traversable which has been placed on the path. This in turn makes rerunning the algorithm potentially cheaper than having to recalculate the entire path.

1.1.2 Description of A star

Astar is a heuristic algorithm being 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.

1.1.3 Description of Dijkstras algorithm

Dijkstras 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, Dijkstras only uses the distance from the start node of each node to find compute the shortest path

1.1.4 Description of Lifelong planning Astar

Lifelong planning Astar is an incremental pathfinding algorithm that finds the path by updating the gcost of nodes from previous searches rather than recalulating the entire graph it is one step down from Dstar Lite which is a continuation of the lifelong planning astar algorithm.

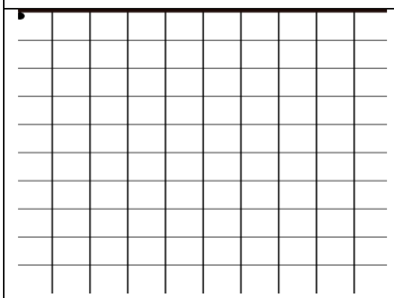
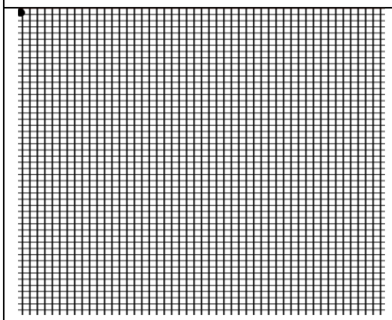
1.1.5 Description of Depth First Search

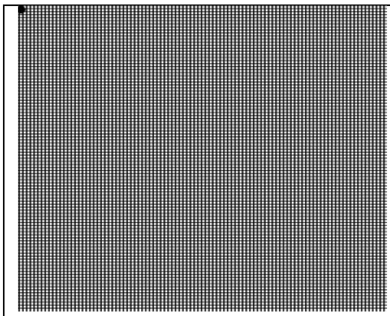
Depth first search is an example of a non- heuristic guided search algorithm, it starts at the root of the graph in the case of this project being the start node which you select. It then traverses through the graph using the neighbours so eventually find the goal. It is not guided it simply goes as far as it can given a specific direction chosen.

2.0 Functional Specification

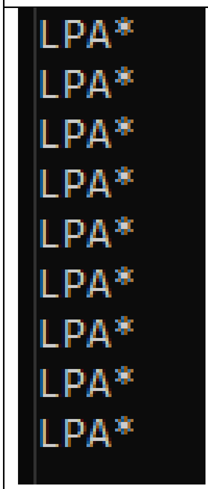
The software will in essence function as a visual pathfinding application. So the user will run the application and see a basic grid they can then adjust the size of the grid to three specified sizes “Small” being a 10x10 grid, “Medium” being a 50x50 grid and “Large” being a 100x100 size grid. They can then choose from a variety of pathfinding algorithms them being Astar, Dstar Lite, Dijkstra’s algorithm, lifelong planning Astar , jump point search and the only no heuristic pathfinding algorithm depth first search. The user can also place down obstacles during process of the algorithms search and before the algorithm has been ran if they perhaps are looking for a specific path, onto the grid which will have the pathfinding algorithms react to them and find a corresponding path.

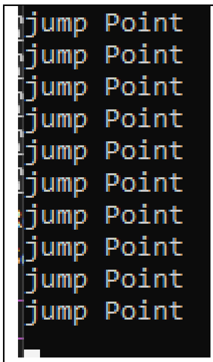
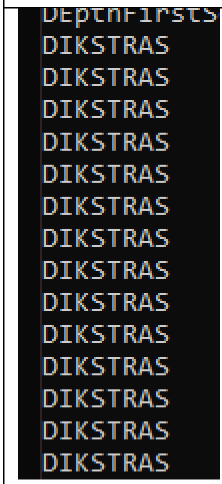
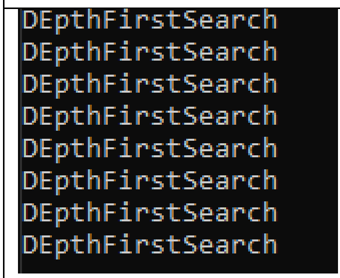
2.1 Algorithm Table and Grid Sizes

	The Menu: The user can select grid size and the algorithm they wish to use.
	Grid size “Small”: The small grid with 100 cells and row of columns of 10 each
	Grid size “Medium”: The medium grid with 2500 cells and row of columns of 50 each

	Grid size “Large”: The large grid with 10,000 cells and row of columns of 100 each

2.2 Console Visualisation: (ScreenShots from the console)

	The Astar Search algorithm being ran and displayed in the console there will be a visual representation in the application.
	The Lifelong Planning Astar (LPA*) algorithm being ran and displayed in the console there will be a visual representation in the application.

	The Jump Point Search algorithm being ran and displayed in the console there will be a visual representation in the application.
	The Dstar Lite Search algorithm being ran and displayed in the console there will be a visual representation in the application.
	The Dijkstra's Search algorithm being ran and displayed in the console there will be a visual representation in the application.
	The Depth First Search algorithm being ran and displayed in the console there will be a visual representation in the application.

2.3 Code visualisation: (code snippets)

```
std::stack<Cell*> Grid::LPAStar(Cell* t_start, Cell* t_goal)
{
    // init the grid to suit cinditions
    sf::Clock m_clock;
    m_LpaStartimer.asSeconds();
    m_LpaStartimer = m_clock.restart();

    if (LPApathFound == false)
    {
        for (int i = 0; i < MAX_CELLS; i++)
        {
            Cell* v = atIndex(i);

            v->setPrev(nullptr);
            v->setHcost(heuristic(v, t_goal));
            v->setMarked(false);
            v->setGcost(M_INFINITY);
            v->setWieght(10);
            v->setKey(M_INFINITY, M_INFINITY );
            if (v->getTraversable() == true)
            {
                v->setColor(sf::Color::White);
            }

        }
    }

    // init start g cost and rhs cost to 0
    t_start->setGcost(0);
    t_start->setRHSCost(0);
    // input start key to suit that

    t_start->setKey(std::min(t_start->getGcost(), t_start->getRhSCost()) + heuristic(t_start, t_goal) + K_M * EPS, 0);

    std::priority_queue<Cell*, std::vector<Cell*>, KeyComparer> u;
    u.push(t_start);
```

The initilise portion of the Lifelong Planning Astar fuction

```
while (!u.empty() && LPApathFound == false)
{

    // Get the top cell from the priority queue
    Cell* curr = u.top();
    curr->isInOpenList = false;
    u.po  (local variable) Cell *curr
          Get the top cell from the priority queue
    // I Search Online                          out
    if (
    {
        std::cout << "Found cell" << std::endl;
        m_LpaStartimer = m_clock.getElapsedTime();
        LPApathFound = true;
        break;
    }

    // if the current cell is not a wall
    if (curr->getTraversable() == true)
    {

        // if curr gcost and rhs cost is greater than its rhs cost
        if (curr->getKey().first > curr->getKey().second)
        {
            // setting the colour of searched cell

            // setting the rhs cost = to the g newly caluclated g cost
            curr->setRHSCost(curr->getGcost());
            // searching the neighbours of te current cell
            for (auto succ : curr->getNeighbours())
            {
                // this is the new cost of which is the g cost and the distance from the successor to reorder the pq
                double new_cost = curr->getGcost() + heuristic(curr, succ);
                if (new_cost < succ->getGcost()) {
                    // checking if it is not a wall
                    if (succ->getTraversable() == true)
                    {
                        // setting its previous for rebuilding the path
                        succ->setPrev(curr);
                        succ->setGcost(new_cost);
                        // setting the rhs of a suitable neighbour to the min of current rhs cost and the distance of the successor and the current cell
                        succ->setRHSCost(std::min(succ->getRhSCost(), curr->getGcost() + heuristic(curr, succ)));
                        // checking if it is in the open list
                        if (succ->isInOpenList) {
                            // updating the node it the function where i recalculate the rhs and g coset
                            updateNode(succ, t_goal);
                        }
```

Compute shortest path function.

Part A

```
                        }
                        else {
                            succ->isInOpenList = true;
                            u.push(succ);
                        }
                    }
                }
            }
        }
        else
        {
            curr->setGcost(M_INFINITY);
            for (auto succ : curr->getNeighbours())
            {
                succ->isInOpenList = true;
                updateNode(succ, t_goal);
                u.push(succ);
            }
            curr->isInOpenList = true;
            updateNode(curr, t_goal);
            u.push(curr);
        }
    }
    else
    {
        curr->setKey(M_INFINITY, M_INFINITY);
    }

}
std::stack<Cell*> pathTVec= std::stack<Cell*>();

if (LPApathFound==true)
{
    // Reconstruct the path from start to goal
    Cell* pathNode = t_goal;

    while (pathNode != nullptr)
    {
        std::cout << pathNode->getID() << std::endl;
        pathTVec.push(pathNode);
        pathNode = pathNode->GetPrev();

    }
    std::cout << " path vec size " << pathTVec.size() <<std::endl;;

}

return pathTVec;
```

Compute shortest path function.

Part B

```cpp
void Grid::updateNode(Cell* node, Cell* t_goal)
{
    if (node->getTraversable() == true)
    {


        if (node->isInOpenList) {
            // Update the key if the node is in the open list k_m is the maximun cost per move allowed  and eps being the is an estimate on the cost to go to the goal

            node->setKey(std::min(node->getGcost(), node->getRhSCost()) + heuristic(node, t_goal) + K_M * EPS,
                std::min(node->getGcost(), node->getRhSCost()));
        }
        else {
            // Add the node to the open list with a key of (infinity, infinity)
            node->setKey(std::min(node->getGcost(), node->getRhSCost()) + heuristic(node, t_goal) + K_M * M_INFINITY,
                std::min(node->getGcost(), node->getRhSCost()));
            node->isInOpenList = true;
            //u.push(node);
        }

        // Update the rhs value of the node
        if (node == t_goal) {
            node->setRHSCost(0);
        }
        else {
            double min_rhs = M_INFINITY;

            for (auto successor : node->getNeighbours()) {


                if (successor->getTraversable() == true)
                {
                    double cost = successor->getGcost() + heuristic(node, successor);
                    if (cost < min_rhs) {
                        min_rhs = cost;
                        //node->setPrev(succ); // Set the "previous" attribute to the best successor
                    }
                }

            }
            node->setRHSCost(min_rhs);
        }
    }


}
```

The update node function which based on whether it is in the queue and if the node is not equal to the goal node

```cpp
std::stack<Cell*> Grid::Djkstras(Cell* t_start,Cell* t_goal) {
    sf::Clock m_clock;
    DjkstrasTimer.asSeconds();
    DjkstrasTimer = m_clock.restart();
    Cell* s = t_start;
    Cell* g = t_goal;
    Cell* child;
    std::priority_queue<Cell*, std::vector<Cell*>,GCostComparer > pq;
    std::stack<Cell*> m_stack;

    for (int i = 0; i < MAX_CELLS; i++)
    {
        Cell* v = atIndex(i);
        v->setPrev(nullptr);
        v->setMarked(false);
        v->setGcost(M_INFINITY);
        v->setWieght(10);
        if (v->getTraversable() == true)
        {
            v->setColor(sf::Color::White);
        }

    }

    // initilise the distance of s to 0
    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);
                }
            }
        }
```

The dijkstras algorithm function which returns the shortest path based of a nodes G cost

Part A

```cpp
        }

        pq.pop();

    }
    Cell* pathNode = t_goal;
    while (pathNode->GetPrev()!=nullptr)
    {
        m_stack.push(pathNode);
        if (pathNode != t_start)
        {
            pathNode->setColor(sf::Color::Black);
        }


        pathNode = pathNode->GetPrev();
    }

    t_goal->setColor(sf::Color::Magenta);

    return m_stack;
}
```

Part B

```cpp
class GCostComparer
{
public:
    bool operator()(Cell* t_n1, Cell* t_n2) const
    {
        return (t_n1->getGcost()) > (t_n2->getGcost());
    }
};
```

The g cost comparer for Dijkstra’s which compares the Gcost of two separate nodes

```cpp
std::stack<Cell*> Grid::depthfirstSearch(Cell* t_curr,Cell* t_goal) {


    sf::Clock m_clock;
    depthfirstSearchTimer.asSeconds();
    if (depthGoalFound== true)
    {
        DjkstrasTimer = m_clock.restart();
    }


    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;
                DjkstrasTimer = m_clock.getElapsedTime();
                depthGoalFound = true;
                break;
            }
            if ((*itr)->getMarked() == false)
            {
                (*itr)->setPrev(t_curr);
                depthfirstSearch((*itr),t_goal);

            }
        }
    }


    Cell* pathNode = t_goal;
    while (pathNode->GetPrev() != nullptr)
    {
        m_stack.push(pathNode);
        pathNode->setColor(sf::Color::Black);
        pathNode = pathNode->GetPrev();
    }

    t_goal->setColor(sf::Color::Magenta);
```

The depth first search function which returns a path by following whatever node is next

```cpp
double heuristic(Cell* c1, Cell* c2)
{
    int dx = abs(c1->Xpos - c2->Xpos);
    int dy = abs(c1->Ypos - c2->Ypos);
    int distance = sqrt(dx * dx + dy * dy);

    return c1->getWeight()*( distance);
}
```

My heuristic function which calculates the distance from a given node to another. It is called by passing through the end node and the current node

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


    double heuristicVal = 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;
}
```

The calculation of the D star key

```cpp
std::stack<Cell*> Grid::DstarLiteMain(Cell* t_start, Cell* t_currentSearch) {


    sf::Clock m_clock;
    dStarLiteTimer.asSeconds();
    dStarLiteTimer = m_clock.restart();

    Cell* s_Last = t_start;
    Cell* goal = t_currentSearch;
    initDstar(t_start,t_currentSearch);
    ComputeShortestPath(t_start,t_currentSearch);


    // t_start . gcost is inifinte still there is no known path
    float tempMin = M_INFINITY;
    Cell* nextNode = nullptr;
    while (t_start!= goal)
    {
        for (auto neighbours : t_start->getNeighbours())
        {

            if (neighbours->getWeight()+neighbours->getGcost()< tempMin)
            {
                tempMin = (neighbours->getGcost() + neighbours->getWeight());
                nextNode = neighbours;
            }
        }
        t_start->setColor(sf::Color::Green);



        t_start = nextNode;
    }

    int q = 0;
    // checking for any edge cost changes of the surrounding neighbours

    for (auto neighbours : t_start->getNeighbours())
    {

        if (neighbours->getTraversable() == false)
        {
            K_M = K_M + heuristic(s_Last, t_start);
            s_Last = t_start;



            updateVertex(neighbours,t_currentSearch);
        }
        ComputeShortestPath(t_start, t_currentSearch);
    }

    if (t_start == goal)
    {
        dStarLiteTimer = m_clock.getElapsedTime();
    }


    std::stack<Cell*> shortestPath;


    return shortestPath;

}
```

The Main Dstar lite function which controls the wall handling and progression of the robot position on the path

```cpp
void Grid::ComputeShortestPath(Cell * t_start,Cell * t_currentSearch)
{

    while (U_pq.top()->getKey() < calculateDstarKey(t_start,t_start) || t_start->getGcost() != t_start->getRhSCost())
    {
        Cell* currentCell = U_pq.top();
        if (currentCell != nullptr)
        {


            std::pair<double, double> key_Old = U_pq.top()->getKey();
            std::pair<double, double> key_New = calculateDstarKey(currentCell, t_start);

            currentCell->setMarked(true);
            U_pq.pop();
            if (currentCell->getTraversable() == false)
            {
                continue;
            }

            if (key_Old < key_New)
            {
                currentCell->setKey(key_New.first, key_New.second);
                U_pq.push(currentCell);
            }// if it is overconsistant
            else if (currentCell->getGcost() > currentCell->getRhSCost())
            {

                // relaxing the node

                currentCell->setGcost(currentCell->getRhSCost());

                for (auto pre : currentCell->getNeighbours())
                {
                    // update their vertexes


                    updateVertex(pre, t_currentSearch);
                }

            }
            else {
                // if the nodeis underconsistant

                currentCell->setGcost(M_INFINITY);

                for (auto neighbours : currentCell->getNeighbours())
                {

                    updateVertex(neighbours, t_currentSearch);

                }

                updateVertex(currentCell, t_currentSearch);
            }
        }
    }




}
```

The compute shortest path function which controls the handling of the different types of inconsistent nodes being over consistent and under consistent.

```cpp
void Grid::initDstar(Cell* t_start , Cell* t_currentSearch)
{
    U_pq = std::priority_queue<Cell*, std::vector<Cell*>, DstarKeyComparer>();
    K_M = 0;

    for (int i=0;i<MAX_CELLS;i++)
    {
        Cell* v = atIndex(i);
        v->setPrev(nullptr);

        // set all gcosets to infinity
        v->setGcost(M_INFINITY);

        v->setRHSCost(M_INFINITY);

        v->setWieght(10);
        v->setMarked(false);
        if (v->getTraversable() == true)
        {
            if (v != t_currentSearch)
            {
                v->setColor(sf::Color::White);
            }

        }
    }


    t_currentSearch->setRHSCost(0);
    t_currentSearch->setKey(calculateDstarKey(t_currentSearch, t_start).first, calculateDstarKey(t_currentSearch, t_start).second);
    U_pq.push(t_currentSearch);

```

The initialise Dstar function initialises all of the nodes in the grid to suit for the algorithm to work and pushes the goal node into the priority queue with the correct RHS cost and Gcost

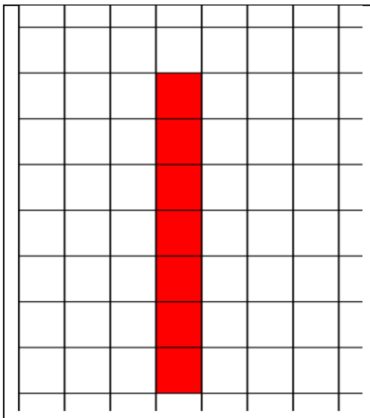
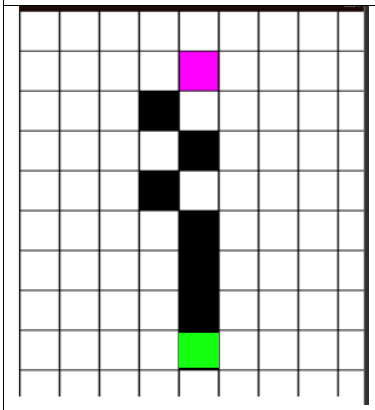
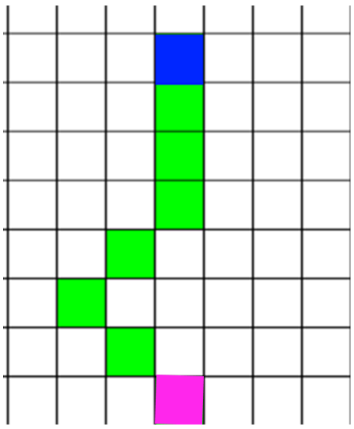
```cpp
void Grid::updateVertex(Cell* currentCell,Cell * t_start) {

    if (currentCell != nullptr|| t_start!= nullptr)
    {
        if (currentCell != t_start)
        {
            double tempMin = M_INFINITY;
            for (auto neighbours : currentCell->getNeighbours())
            {
                if (neighbours->getGcost() + neighbours->getWeight() < tempMin)
                {

                    tempMin = (neighbours->getGcost() + neighbours->getWeight());
                }
            }

            currentCell->setRHSCost(tempMin);
        }


        priority_queue<Cell*> new_pq;
        while (!U_pq.empty()) {
            Cell* current = U_pq.top();
            U_pq.pop();
            if (current != currentCell) {
                new_pq.push(current);
            }

        }

        while (!new_pq.empty())
        {
            U_pq.push(new_pq.top());
            new_pq.pop();
        }

        if (currentCell != nullptr)
        {
            if (currentCell->getMarked() == false)
            {


                if (currentCell->getGcost() != currentCell->getRhSCost())
                {


                    currentCell->setKey(calculateDstarKey(currentCell, t_start).first, calculateDstarKey(currentCell, t_start).second);
                    U_pq.push(currentCell);
                }
            }
        }
    }

}
```

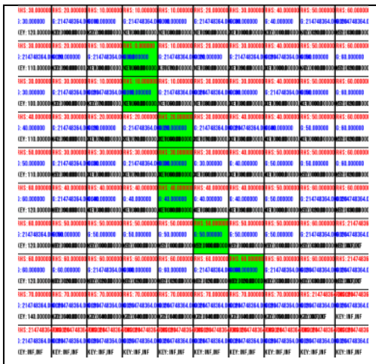
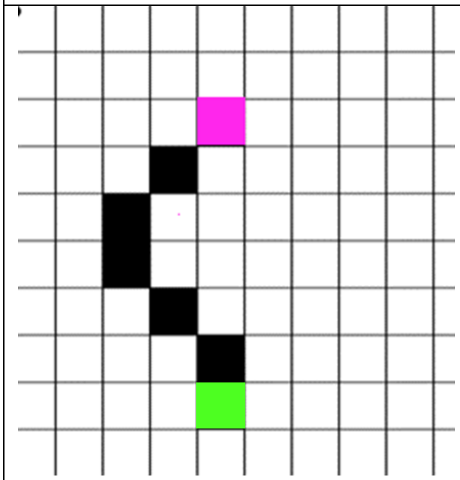
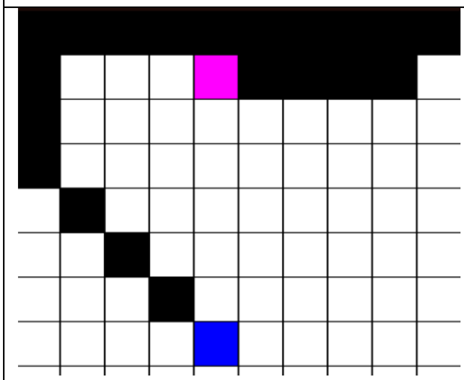
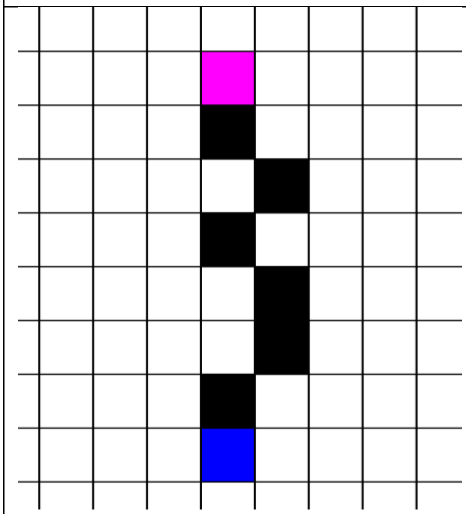
The Dstar Lite Update Vertex function which updates the key costs of nodes and potentially pushes them into the queue for further investigation.

```cpp
Cell* start = t_start;
Cell* goal = t_end;
std::priority_queue<Cell*, std::vector<Cell*>, CostDistanceValueComparer > pq;
pq.empty();
m_stack.empty();
t_path.clear();
int infinity = std::numeric_limits<int>::max() / 10;

for (int i = 0; i < MAX_CELLS; i++)
{
    Cell* v = atIndex(i);
    v->setPrev(nullptr);
    v->setHcost(abs(goal->xPos - v->xPos) + abs(goal->yPos - v->yPos));
    v->setMarked(false);
    //v->setRHSCost(abs(v->GetPrev()->getHcost() + v->getHcost()));
    v->setGcost( infinity);
    v->setWieght(10);
}

start->setGcost(0);

pq.push(start);

pq.top()->setMarked(true);

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->getMarked() == false)
            {
                pq.push(child);
                child->setMarked(true);
            }
            // this will run astar around the intraversable but it doesnt fix anyhting
            /*if (distanceToChild < child->getGcost() && child->getTraversable() == fa
            {
                Dstar(child, goal);
            }*/
        }
    }

    pq.pop();
}


if (m_status == false)
{
    Cell* pathNode = t_end;

    while (pathNode->GetPrev() != nullptr)
    {
        t_path.push_back(pathNode->getID());
        pathNode = pathNode->GetPrev();
        m_stack.push(pathNode);
    }

    for (int i = 0; i < t_path.size(); i++)
    {
        Cell* m = atIndex(t_path.at(i));
        m->getRect().setFillColor(sf::Color::Black);
    }
}


return m_stack;
```

My Astar function compute shortest path function

2.4 Visualisation of Paths using the different algorithms available.

	an example of basic walls. Red Nodes are the walls placed on the grid. The grid size in question is the small grid of size 100 nodes.
	Path returned using Astar Search algorithm on a grid size "Small" Green Node = Start Node Magenta Node = Goal Node
	Path returned using Dstar Lite Search algorithm on a grid size "Small" Magenta Node = Start Node Blue Node = Goal Node Dstar Lite with Debug on second Screen

	
	Path returned using Lifelong Planning Astar Search algorithm on a grid size "Small" Green Node = Start Node Magenta Node = Goal Node
	Path returned using Depth First Search algorithm on a grid size "Small"
	Path returned using Dijkstra's Search algorithm on a grid size "Small"

2.5 Data Collection visualisation

small0.090104 small0.090104 small0.090104 small0.090104 small0.090104 small0.035049 small0.035049 small0.035049 small0.035049 small0.035049	Dstar lite Small grid data
medium0.607782 medium0.607782 medium0.587208 medium0.587208 medium0.587208 medium0.587208 medium0.587208 medium0.587208 medium0.587208	Dstar lite medium grid data
4 large30.132551 5 large30.132551 6 large30.132551 7 large30.132551 8 large30.132551 9 large30.132551 0 large15.718164 1 large15.718164	Dstar lite large grid data

0.138869 0.135377 0.137532 0.138712 0.137757 0.13468 0.13875 0.134387 0.139464 0.14005 0.135846 0.13409	Astar small grid data
0.054926 0.052687 0.05154 0.053191 0.053235 0.053147 0.052287 0.057563 0.06216 0.053713 0.05319 0.05194	Astar medium grid data
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	Astar large grid data

small0.004657 small0.004215 small0.004081 small0.004760 small0.004344 small0.004281 small0.004179 small0.004119 small0.004166 small0.004177 small0.004104	Dijkstra’s small grid data
medium0.106067 medium0.101982 medium0.100599 medium0.099943 medium0.103269 medium0.104548 medium0.101578 medium0.107218 medium0.103903	Dijkstra’s medium grid data
large0.306492 large0.314421 large0.310135 large0.305354 large0.310717 large0.311183 large0.320436 large0.301858 large0.311105 large0.302993 large0.310332 large0.302764 large0.313947 large0.303118 large0.304529 large0.314564 large0.306293 large0.310978 large0.394907 large0.415434	Dijkstra’s large grid data

SMALL0.019829 SMALL0.019031 SMALL0.057758	Lifelong Planning Astar large grid data
MEDIUM1.035708 MEDIUM1.051293	Lifelong Planning Astar medium grid data
large5.850010 large35.807903	Lifelong Planning Astar large grid data

3.0 Design and describe how the application will be used:

The user will be met with a screen 1 of the 3 screens. They are given the option to choose the size of the grid which they want to place the algorithm on and the algorithm itself which they want to use. They can choose from each of the algorithms and change them whenever they want by simply just selecting another algorithm.The user can then select whether or not they want to race the algorithms. If they select the option to do so another screen will appear where their start and end poisitions chosen will be mirrored. They will then see Dstar Lite only race against their chosen algorithm. They can then choose whether or not to turn on the debug option what this will do is allow the user to see the individual costs of cells such as their “RHS” (Right hand side) cost their “Gcost”(distance from the start position) and their key values (a calculation necessary to Dstar Lite) on each individual cell and will see these values change as the algorithm progresses. They toggle this on and off as they wish. This debug visualisation is only available on the small sized grid as its to obscure to see as the grid size increases.However they can race the algorithms on each grid size.

4.0 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