Computer Games Development SE607
Technical Design Document
Year IV

Donal Howe
C00249662

[Student Name]
[Student Number]

[Date of Submission]

[Declaration form to be attached]

Table of contents

1. Introduction

The objective of this project is to compare the benefits and drawbacks of using commonly used heuristic based guided pathfinding algorithms to the incremental algorithm known as Dstar Lite. This project will discuss the direct benefits of each algorithm in depth, from Astar, Dijkstras search algorithm, Lifelong Planning Astar, and the non-guided algorithm known as Depth First Search when compared to D star Lite within a game's context.

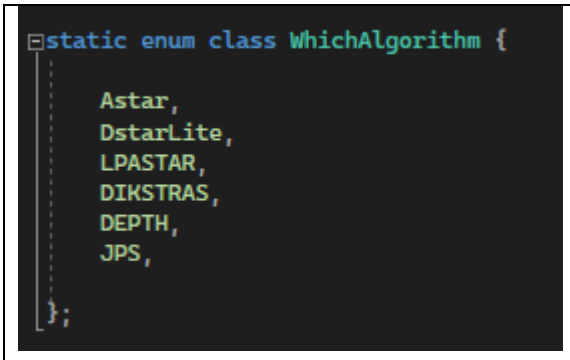
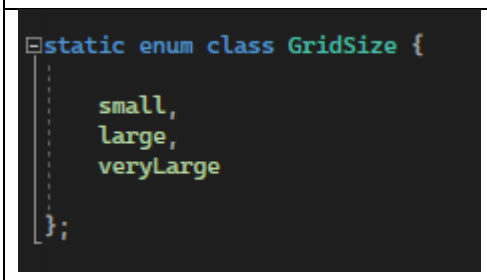
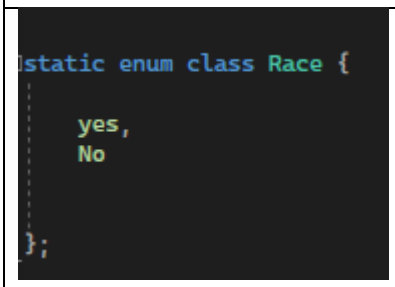
2. Technical Design

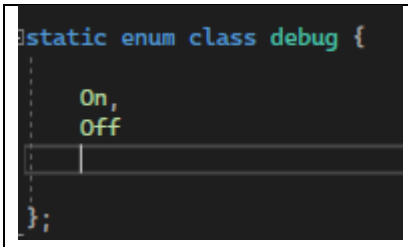
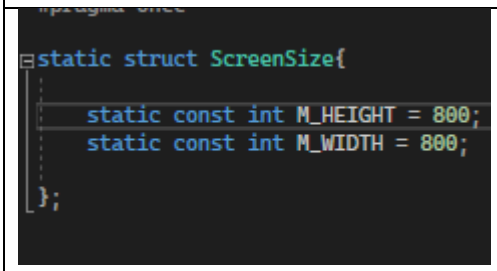
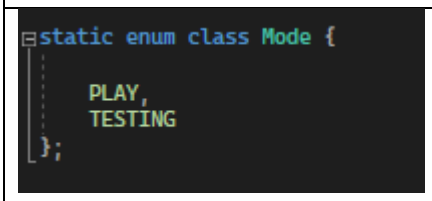
The purpose of this document is to effectively communicate the technical details and design decisions of the system/algorithm to the readers.

It could include software architecture, algorithm design, class specifications, pseudo code, etc. with tools such as UML, Class Diagram, CRC Cards.

2 .1 Class Snippets: Header File

Helper classes, Structs:

	The Enum class called “WhichAlgorithm” which controls which algorithm is being used a certain time. This Enum class contains the name for every pathfinding algorithm in the project.
	The Enum class called “GridSize” controls the size of the grid which is being used in the program ranging from “small” to “very Large”. “small” = “10x10” grid “Large” “50x50” grid “very Large” = “100x100” grid
	This Enum class called “Race” depicts whether you want to race the algorithms in comparison to dstar lite on a chosen path

	The Enum class called “debug” toggles whether the user wants to see the variable values for Dstar Lite on the screen. This is only available with the small grid size
	The struct called “Screen Size” struct which controls the size of each window
	The Enum class called “Mode” which controls which mode the application is in. behaving differently depending on which one it is in

2.2 Utilised Class Snippets:

<table>
<tr>
<td>

```
class Cell
{
public:
    Cell();
    ~Cell();
    void render(sf::RenderWindow& t_window);
    void setStartColour();
    void setEndColour();
    void setColor(sf::Color t_color);
    void setMarked(bool t_marked);
    bool& getMarked();
    void setTraversable(bool t_traversable);
    bool& getTraversable();
    void setEndPoint(bool t_isEndpoint);
    void setStartPoint(bool t_isStartpoint);
    int& getID();
    void setID(int t_id);
    double& getGcost();
    void setGcost(double t_gcost);
    double& getHcost();
    void setHcost(double t_hcost);
    double& getRHSCost();
    void setRHSCost(double t_rhs);
    bool& GetRisenBool();
    void setRisenBool(bool t_isRisen);
    void raiseCost(double t_raise);
    void setWieght(int t_w);
    int& getWeight();
    sf::Vector2f& getPos();
    void setPos(sf::Vector2f t_pos);
    sf::RectangleShape& getRect();
    void initRect(int t_c);
    Cell* GetPrev();
    void setPrev(Cell* t_prev);
    std::list<Cell*>& getNeighbours();
    void setNeighbours(Cell* t_neighbour);
    std::list<Cell*>& getPredecessors();
    void setPredecessorss(Cell* t_neighbour);
    void setFcost(double t_fcost);
    double& getFcost();
    bool &getJumpPoint();
    void setJumpPoint(bool t_b);
    double m_Gcost=0;
    double m_Hcost;
    double m_RHScost;
    double m_Fcost;
    float Xpos;
    float Ypos;
    int m_wieght;
    bool isInOpenList = false;
    std::pair<double, double> &getKey();
    void setKey(double t1,double t2);
    std::pair<double,double> m_key;
    sf::Text m_rhsText;
    sf::Text m_GcostText;
    sf::Text m_KeyText;
```

</td>
<td>Public variables of Cell Class

the header file for the Cell(node) which has all the current functions in use.</td>
</tr>
<tr>
<td>

```
private:
    bool m_marked;
    bool m_isEndoint;
    bool m_isStartoint;
    bool m_traversable;
    bool isJumpPoint = false;
    int m_ID;
    bool m_HcostRisen;
    bool m_HcostLowered;
    sf::Vector2f m_pos;
    sf::RectangleShape m_rect;
    Cell* prev;
    std::list<Cell*> m_neighbour;
    std::list<Cell*> m_predecessors;
```

</td>
<td>Private variables to Cell Class</td>
</tr>
</table>

```
class Grid
{

    sf::Font m_font;

    // just used for cell setup in grid
    Cell *ptrCell;

    // grid size values
    int MAX_CELLS;
    int numberOfRows;
    int numberOfCols;
```

private member variables of the grid class

```
public:
    Grid();
    ~Grid();

    void setMAXCELLS( int t_cellCount);
    void setColumns( int t_ColCount);
    void setRows( int t_rowCount);

    int& getMAXCELLS();
    int& getNumberOfRows();
    int& getnumberOfCols();

    // function that uses the id of a cell to return a ptr to the actual cell
    Cell* atIndex(int t_id);

    // the grid itself
    std::vector<std::vector<Cell>> m_theTableVector;

    // if start and endpoints fo algorithms are chosen these two are modifies
    bool m_startPosChosen = false;
    bool m_endPosChosen = false;

    const double M_INFINITY = std::numeric_limits<int>::max() / 10;

    // sets the neigbours/successors of a cell
    void setNeighbours(Cell* t_cell);

    // sets the predeeccessors of a cell
    void setPredecessors(Cell* t_cell);

    // sets up the grid and neccessary values for cells
    void setupGrid(int t_count);

    void render(sf::RenderWindow & t_window);

    //calculates the heuristic  value of the the cells inputed
    double heuristic(Cell* c1, Cell* c2);
```

public members of the Grid class

2.3, 2.4 Algorithm's Classes: + UI and Game Class

```
/// <summary>
/// compares the first key against the second to return the smallest
/// if there is a tie between the tow, it will return the higher in the priority queue
/// </summary>
///
class DstarKeyComparer {
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;
        }
    }
};
```

Functor used in Dstar Lite

```
class DstarLite
{
    //k_m = key modifier
    // it accounts for the moving of the start node which in turn would change the heuristic of further
    // away nodes  if this did not account for that change
    float K_M;

    // timer for dstar
    sf::Time dStarLiteTimer;

    // termination condition
    bool dstarGoalFound = false;
```

Private members of the Dstar Lite Class

```
// returns the timer for DFS
sf::Time& getTimer();

// returns the termination condition
bool& getDStarPathFound();

// priority queue  tracks nodes which are being investigated
std::priority_queue<Cell*, std::vector<Cell*>, DstarKeyComparer> U_pq;

// the main function which handles moving of the start node and obstacle handling
void DstarLiteMain(Cell* t_finalGoal, Cell* t_StartCurr,Grid * t_grid);

// updates the costs of each node accordingly depending on the type of inconsistancy
void updateVertex(Cell* currentCell, Cell* t_finalGoal,Grid * t_grid);

// computes the shortest path and checks what type of inconsistancy is the node or if it is consistant
void ComputeShortestPath(Cell* t_start, Cell* t_StartCurrGrid,Grid * t_grid);

// initilises the variables for dstar
void initDstar(Cell* t_finalGoal, Cell* t_StartCurr,Grid * t_grid);

// calculates the key for dstar
std::pair<double, double> calculateDstarKey(Cell* t_StartCurr, Cell* t_finalGoal,Grid * t_grid);

// s_last used to keep track of robots position on the grid
Cell* s_Last;
```

Public members of the Dstar Lite Class

```
/// <summary>
/// compares the fcost  of cell 1 against cell 2's f cost to return the lower of the tmo
/// this functor is used for astar to return the better f cost
/// </summary>
class CostDistanceValueComparer
{

public:

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

Functor used in “Astar”.

```
class Astar
{
    //astar
    // this is the timer used to calculate the time until completion of the algorithm
    sf::Time m_Astartimer;
    // bool to control if the algorithm is done
    bool AstarDone = false;

public:
    // returns the timer
    sf::Time& getTimer();

    //returns the termination condition
    bool &getIfDone();

    // initilises the astar grid
    void AstarInit(Cell* t_finalGoal, Cell* t_StartCurr, Grid* t_grid);
    // computes the shortestPath for the astar search
     std::stack<Cell*> computeShortestPath(Cell*  t_start, Cell * t_goal,Grid* t_grid);

    std::stack<Cell*> m_stack;
    // contructor
    Astar();
    //destructor
    ~Astar();
};
```

“Astar” class as declared in the header file

```
/// <summary>
/// compares the first key against the second to return the smallest
/// if there is a tie between the two it it returns the higher in the priority queue
/// </summary>
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;
        }
    }
};
```

Functor used in “LpaStar”

<table>
<tr>
<td>

```
class LpaStar
{
    // the clock for timer
    sf::Clock m_clock;

    //k_m is the maximun cost per move allowedand eps being the is an estimate on the cost to go to the goal
    const float EPS = 2.0f;

    // k_m is the key modifier a value that changes as the search progresses
    float K_M;

    // the timer for lpa star tracks timer to completion
    sf::Time m_LpaStartimer;

    // temination condition
    bool LPApathFound = false;

public:

    // sets the bool for temination back to false
    void setTerminationCondition(bool t_bool);

    // returns the timer for DFS
    sf::Time& getTimer();

    // returns the termination condition
    bool& getLpaStarPathFound();

    //initilises variables for lpaStar
    void initLpaStar(Cell* t_start, Cell* t_goal, Grid* t_grid);

    // lpa star function finds the path
    void LPAStar(Cell* t_start, Cell* t_goal, Grid* t_grid);

    // update the values of each node or vertex
    void updateNode(Cell* node, Cell* Goal, Grid* t_grid);

    //calculates the key of each node
    std::pair<double, double> calculateKey(Cell* s, Cell* t_goal, Grid* t_grid);

    // default constructor
    LpaStar();

    // default destructor
    ~LpaStar();
};
```

</td>
<td>“LpaStar” class as declared in the header file</td>
</tr>
<tr>
<td>

```
/// <summary>
/// compares the Gcost  of cell 1 against cell 2's Gcost to return the lower of the two
/// this functor is used for dijkstras search  to return the lower g cost
/// </summary>

class GCostComparer
{

public:

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

</td>
<td>Functor used in “Dijkstra’s” search algorithm.</td>
</tr>
<tr>
<td>

```
class Dijkstras
{
    bool djkstrasPathFound = false;
    sf::Time DjkstrasTimer;

public:

    // returns the timer for DFS
    sf::Time& getTimer();

    // returns the termination condition
    bool& getDijkstrasPathFound();

    void computeShortestPath(Cell* t_start, Cell* t_Goal, Grid* t_grid);

    Dijkstras();
    ~Dijkstras();
};
```

</td>
<td>“Dijkstra’s” Class as declared in the header file</td>
</tr>
<tr>
<td>

```
#include "Cell.h"
class DepthFirstSearch
{
    bool depthGoalFound = false;

    sf::Time depthfirstSearchTimer;

public:

    // returns the timer for DFS
    sf::Time& getTimer();

    // returns the termination condition
    bool& getDepthFound();

    // computes the path for depth first search
    void computeShortestPath(Cell* t_curr, Cell* t_goal, Grid* t_grid);

    // constructor
    DepthFirstSearch();

    // destructor
    ~DepthFirstSearch();
};
```

</td>
<td>“Depth First Search” Class as declared in the header file.</td>
</tr>
</table>

<table>
<tr>
<td>

```cpp
#include "SFML/Graphics.hpp"
#include <iostream>
#include <vector>
#include "algorithmSwitcher.h"
#include "Grid.h"
#include "Cell.h"
class Menu
{
    GridSize m_gridSwitcher;
    WhichAlgorithm m_slgSwitcher;
    Race m_raceDecider=Race::No;
    debug m_debugDecider=debug::Off;

    sf::RectangleShape m_rect;
    std::vector<sf::RectangleShape> m_rectVec;

    sf::Font m_font;
    sf::Text m_text[13];


public:
    Menu();
    ~Menu();

    WhichAlgorithm& getalg();

    Race& getRaceStatus();
    debug& getdebugStatus();
    std::vector<sf::RectangleShape> getVec();

    GridSize& setGridSize(sf::RenderWindow & t_windowTwo, Grid & t_grid, Grid& t_gridTwo, Cell *t_cell);

    void render(sf::RenderWindow& t_window);
    void update(sf::Time t_deltaTime);
};
```

</td>
<td>“Menu” class as declared in the header file.</td>
</tr>
<tr>
<td>

```cpp
class Game
{

public:

    Game();
    ~Game();

    WhichAlgorithm m_switcher;
    GridSize m_gridSizeState;
    Race m_raceState=Race::No;
    debug m_debugState=debug::Off;
    Astar m_astar;
    Dijkstras m_dijkstras;
    DepthFirstSearch m_depthFirstSearch;
    DstarLite m_dStarLite;
    LpaStar m_LpaStar;

    void run();

private:
```

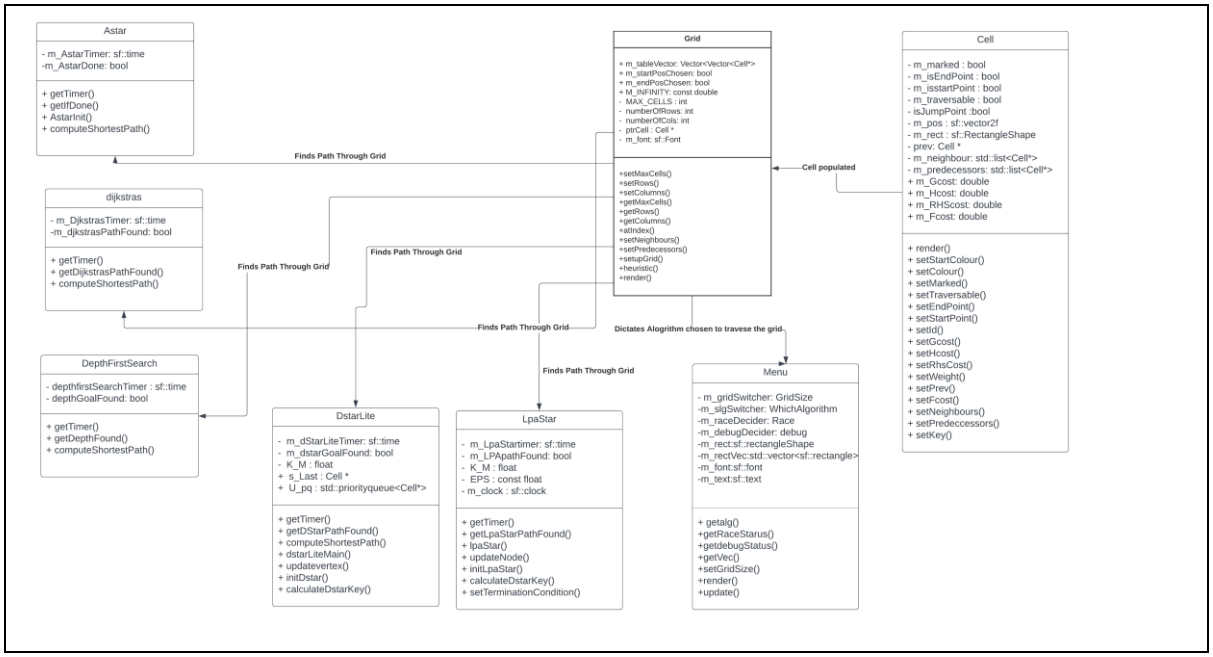
</td>
<td>Public members to class “Game”, as declared in the header file.</td>
</tr>
<tr>
<td>

```cpp
private:
    void PlayMode();
    void TestingMode();
    bool SrtChosen = false;
    bool EndChosen = false;
    bool m_exitGame;
    Cell* tempsEnd;
    Cell* tempstart;
    Cell* tempstartTwo;
    Cell* tempsEndTwo;
    Cell *m_cellVAR;
    std::stack<Cell*> AstarStack;
    Menu m_menu;
    Grid m_grid;
    Grid m_gridTwo;
    sf::RenderWindow m_window;
    sf::RenderWindow m_windowTwo;
    sf::RenderWindow m_windowAstar;
    ofstream outputData;
    Mode m_mode=Mode::PLAY;
    int startCell;
    int startCellTwo;
    int EndCell;
    int EndCellTwo;
    void processEvents();
    void processKeys(sf::Event t_event);
    void processMouseInput(sf::Event t_event);
    void update(sf::Time t_deltaTime);
    void render();

};
```

</td>
<td>Private members to class “Game”, as declared in the header file.</td>
</tr>
</table>

3. Class Diagram:



4. CRC Cards:

Grid	
• Controls the size of grid, and holding onto information of the cells within the grid	• Cell

Game	
• controls the game loop and use of an algorithm	• WhichAlgorithm • GridSize • Race • Debug • Astar • Dijkstras • DepthFirstSearch • DstarLite • LpaStar • Cell • Grid • Mode

Menu	
• controls the selection of each algorithm • Controls the decision to race each algorithm • Controls the decision to see debug options on algorithms	• WhichAlgorithm • Race • Debug • GridSize

Astar	
• Computes the shortest path using the Astar algorithm and returns the path for the robot to follow	• Cell • Grid

DstarLite	
• calculates the optimal path from two chosen points on the grid and returns the path for the robot to follow	• Cell • Grid

Dijkstras	
• calculates the optimal path between two points on the grid and returns the path for the robot to follow	• Cell • Grid

DepthFirstSearch	
• finds a path to the goal from two chosen points on the grid	• Cell • Grid

LpaStar	
• calculates the optimal path from two chosen points on the grid and returns the path for the robot to follow	• Cell • Grid

Cell	
• instantiates the Cell with all of the necessary values required • returns all of the values required	

ScreenSize	
• controls the size of the screen using public const int variables	

Mode	
• an enum class which depicts which mode the application is in	

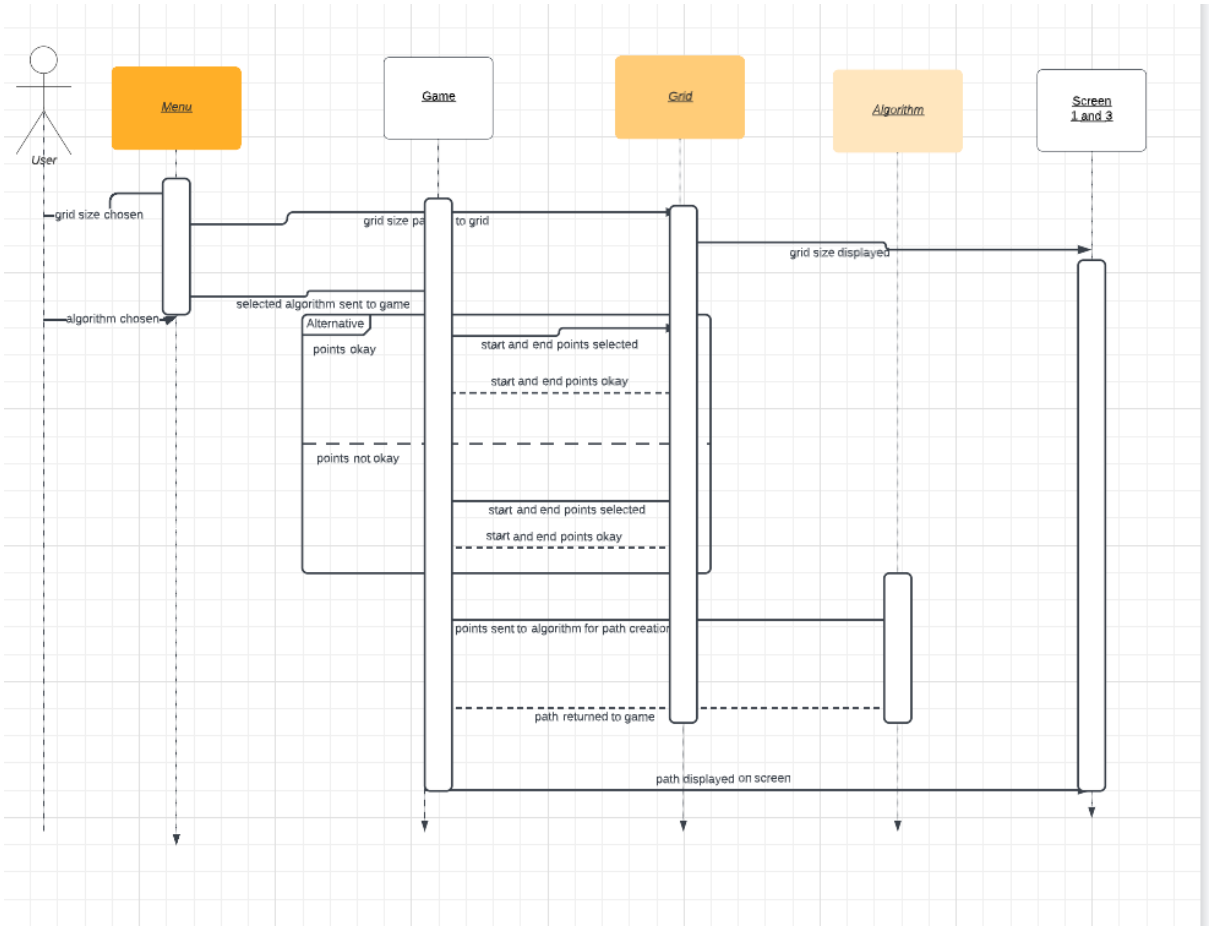
debug	
• enum class which controls is the application is in debug mode or not	

Race	
• enum class which controls is the algorithms are going to race or not	

GridSize	
• enum class which controls the size of the grid	

WhichAlgorithm	
• enum class which controls what algorithm is being used to find a path from two chosen points on the grid	

5.0 Sequence diagram



6.0 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