Computer Games Development
Technical Design Document

Year IV

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

DECLARATION

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

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

Student Name: (Printed) Donal Howe

Student Number(s): C00249662

Signature(s): Donal Howe

Date: 24/04/2023

Please note:

a) * Individual declaration is required by each student for joint projects.
b) Where projects are submitted electronically, students are required to submit their student number.
c) The Institute regulations on plagiarism are set out in Section 10 of Examination and Assessment Regulations published each year in the Student Handbook.

Table Of Contents

List Of Figures

List Of Tables

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, Dijkstra's 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 Introduction

This section shows the technical design and over all architecture of the code including, the enums and structs that were used to control the different algorithms and grid sizes being run in the application. The header files of each algorithm and the rest of the files in the game. It will discuss what each variable and function is and does.

2.2 Header Files

Helper classes and structs code images	Description

`static enum class WhichAlgorithm {`
`    Astar,`
`    DstarLite,`
`    LPASTAR,`
`    DIKSTRAS,`
`    DEPTH,`
`    JPS,`
`};`
	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.

`static enum class GridSize {`
`    small,`
`    large,`
`    veryLarge`
`};`
	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

`static enum class Race {`
`    yes,`
`    No`
`};`
	This Enum class called “Race” depicts whether you want to race the algorithms in comparison to dstar lite on a chosen path

```cpp
static enum class debug {

    On,
    Off

};
```

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

```cpp
static struct ScreenSize{

    static const int M_HEIGHT = 800;
    static const int M_WIDTH = 800;

};
```

The struct called “Screen Size” struct which controls the size of each window

```cpp
static enum class Mode {

    PLAY,
    TESTING
};
```

The Enum class called “Mode” which controls which mode the application is in. behaving differently depending on which one it is in

Table 2-1 Description of The Helper classes and structs code.

Utilised class images	Description

```cpp
// the cell class
class Cell
{
    // public variables of the class
public:

    // the  default constructor
     Cell();

     // default destructor
    ~Cell();

    // sets a the coulour for the first cell in the path
    // where the search starts from
    void setStartColour();

    // sets a the coulour for the last cell in the path
   // where the search end from
    void setEndColour();

    //sets the colour of a cells sf::rectangleShape
    void setColor(sf::Color t_color);

    // sets the cell to have been marked or not
    void setMarked(bool t_marked);

    // returns if the cell has been marked
    bool& getMarked();

    // sets the cells traversable boolean
    void setTraversable(bool t_traversable);

    //returns the cells traversable boolean
    bool& getTraversable();
```

Public variables of Cell Class

The header file for the Cell(node) which has all the current functions in use.

Part A

Cell() = default constructor

~Cell() =default destructor

setStartColour()=sets the start cell of the paths colour

setEndColour()=sets the end cell of the paths colour

setColour()= sets the colour of the cell

Set/GetMarked()= sets and gets the m_marked bool

Set/GetTraversable() = sets and gets the m_traversable bool

```cpp
// sets if the cell is the end poinht in the search
void setEndPoint(bool t_isEndpoint);

// returns the cells enpoint boolean
bool getEndPoint();

// sets if the cell is the start poinht in the search
void setStartPoint(bool t_isStartpoint);

// returns the cells startpoint boolean
bool getStartPoint();

// returns the cells id number
int& getID();

// sets the cells if number
void setID(int t_id);

// returns the cells g cost value
double& getGcost();

// sets the cells g cost value
void setGcost(double t_gcost);

// returns the cells h cost value
double& getHcost();

// sets the cells h cost value
void setHcost(double t_hcost);

// returns the cells rhs cost value
double& getRhSCost();

// sets the cells rhs cost value
void setRHSCost(double t_rhs);
```

Public variables of Cell Class

the header file for the Cell(node) which has all the current functions in use.

Part B

Set/Get EndPoint() = sets and gets the endpoint bool

Set/Get StartPoint() = sets and gets the endpoint bool

Set/Get ID= sets and gets the m_ID for the cell

Set/Get Gcost() = sets and gets the gcost value for the cell

Set/Get Hcost() = sets and gets the Hcost value for the cell

Set/Get Rhscost() = sets and gets the Rhscost value for the cell

```cpp
// returns a cells risen bool which is if its hcost value has been raised
bool& GetRisenBool();

// sets is risen booloan of the cell
void setRisenBool(bool t_isRisen);

// raises the cost of cell
void raiseCost(double t_raise);

// sets the weight of a cell or its edgecost how much it is to move there
void setWieght(int t_w);

// returns a cells weight
int& getWeight();

// returns the cells sf::Rectangle shapes position
sf::Vector2f& getPos();

// sets the position of the cells sf::rectangleShape
void setPos(sf::Vector2f t_pos);

// returns the cells sf::RectangleShape
sf::RectangleShape& getRect();

// fucntion whic creates the sf::rectangle shape
void initRect(int t_c);

// returns a cells previous pointer cell or pointer to that cells parent
Cell* GetPrev();

// sets that cells previous pointer
void setPrev(Cell* t_prev);

// return that cells neighbours
std::list<Cell*>& getNeighbours();
```

Public variables of Cell Class

the header file for the Cell(node) which has all the current functions in use.

Part C

Set/Get RisenBool() =sets and gets the isRisen bool

raiseCost() = raises the hcost of a cell

Set/Get Weight()= sets and gets the weight of a cell

Set/Get getPos() = sets and gets the position of a cell

getRect() =returns the rectangleShape of the cell

initRect()= initialises the rectangle shape and text

Set/Get Prev() = sets and gets the parent pointer of a cell

getNeighbours()= returns the list that stores the neighbours of that cell

```cpp
// sets that cells neigbours
void setNeighbours(Cell* t_neighbour);

// retuns that cells predecessors
std::list<Cell*>& getPredecessors();

// sets the predecessors for that cell
void setPredecessorss(Cell* t_neighbour);

// sets the fcost value for that cell
void setFcost(double t_fcost);

// returns the cell s f cost value
double& getFcost();

// returns if that cell is a jumppoint
bool &getJumpPoint();

// sets if that cell us a jumppoint
void setJumpPoint(bool t_b);

// the cells gcost value
double m_Gcost=0;

// the cells hcost value
double m_Hcost;

// the cells rhscost value
double m_RHScost;

// the cells fcost value
double m_Fcost;

// the cells xpos value which is column number
float m_Xpos;
```

Public variables of Cell Class

the header file for the Cell(node) which has all the current functions in use.

Part D

setNeighbours()= stores the neighbours of the cell into a list

Set/Get Predecessors() = sets and gets the predecessors of the cell

Set/Get Fcost() = sets and gets the fcost value of a cell

Set/Get JumpPoint() = sets and gets the m_isjumpPoint Boolean

m_Gcost= gcost value

m_Hcost= hcost value

m_RHScost= rhscost value

m_Fcost= fcost value

m_Xpos= the x position of the cell

```cpp
// the cells ypos value which is row number
float m_Ypos;

// the cells weight
int m_wieght;

// the cells is in open list boolean
bool m_isInOpenList = false;

// returns the cells std::pair whihc is its key
std::pair<double, double> &getKey();

// sets the values for the cells key
void setKey(double t1,double t2);

// the cells key value
std::pair<double,double> m_key;

// rhs cost text
sf::Text m_rhsText;

// g cost text
sf::Text m_GcostText;

// key cost text
sf::Text m_KeyText;
```

Public variables of Cell Class

the header file for the Cell(node) which has all the current functions in use.

Part E

m_Ypos= the y position of the cell in the grid

m_weight= the cost to move to that cell

m_isInOpenList= bool to check if that cell is in the open list

SetKey()/GetKey() = sets and gets the key value for the cell

m_key= holds the key value for the cell

m_rhsText= is the rhs value in text

m_gcostText= is the gCostvalue in text

m_keyText= is the Key value in text

```cpp
    // privare variables of the class
private:

    // class private booleans
        // if the cell has been visited
    bool m_marked;
        // if the cell is the end desitantion or goal
    bool m_isEndoint;
        // if the cell is the start point
    bool m_isStartoint;
        // if the cell can be traversed
    bool m_traversable;
        // if the cell is a jumppoint for jps search
    bool m_isJumpPoint = false;
        // if the cells hcost has been risen
    bool m_HcostRisen;
    // if the cells hcost has been lowered
    bool m_HcostLowered;

    // class private ints
    int m_ID;

    // class private sf::Vector2f
    sf::Vector2f m_pos;

    // class private sf::rectangles shapes
    sf::RectangleShape m_rect;

    // class private Cell pointers
    Cell* m_prev;

    // class private lists/ datastructures
        // list holds the neighbours/successors of the cell
    std::list<Cell*> m_neighbour;
        // list holds the predecessors of the cell
    std::list<Cell*> m_predecessors;

};
```

Private variables to Cell Class

m_marked= used to check if the cell has been visited

m_Endpoint= used to check if the cell is the end goal of a path

m_isStartPoint= used to check if a cell is a start point of a path

m_traversable= check to see if the cell can be traversed

m_jumpPoint= check to see if the cell is a jumPoint

m_HcostRisen= check to see if the cells hcost has risen

m_HcostLowered= check to see if the cells hcost has been lowered

m_ID = holds the Id of the cell in the grid

m_pos= holds the position of the cell in the grid

m_rect= the sf::rectangleShape of the cell

m_prev = the parent pointer to the cell

m_neighbour = the list of neighbours of that cell

m_predecessors = the list of predecessors of that cell

```cpp
class Grid
{
    // private variavles of the class
private:
    // font for the debug
    sf::Font m_font;

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

    // grid size values
    int m_maxCells;

    // the number of rows in the grid
    int m_numberOfRows;
    int m_numberOfCols;
```

private member variables of the grid class

m_font= font used for cells

m_ptrCell = used in the grid setup

m_maxCells = stores the max possible cells

m_numberOfRows = stores the number of rows for the grid

m_numberOfCols = stores the number of columns for the grid

```cpp
    // public variables of the class
public:
    // default constructor
    Grid();

    // default destructor
    ~Grid();

    // sets the max number of cells allowed in the grid
    void setMAXCELLS( int t_cellCount);

    // sets the max number of columns allowed in the grid
    void setColumns( int t_ColCount);

    // sets the max number of rows allowed in the grid
    void setRows( int t_rowCount);

    // resets the grid in transition
    void resetGrid();

    // resets the grid so the algorithms can be run
    void resetAlgorithm();

    // returns the max number if cells allowed in the grid
    int& getMAXCELLS();

    // returns the max number if rows allowed in the grid
    int& getNumberOfRows();

    // returns the max number if columns allowed in the grid
    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;
```

public members of the Grid class

Part A

Grid() = default constructor

~Grid() = default destructor

setMAXCELLS()= sets the maximum cells of the grid

setColumns()= sets the number of columns of the grid

setRows()= sets the number of rows of the grid

resetGrid() = resets the grid for transition down in grid sizes

resetAlgorithm() = resets the grid so a new algorithm can be run on the grid

atIndex() = gets the cell on the grid using the x and y position

m_tableVector = the two-dimensional grid which holds the grid

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

    // value for infinity
    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);

    // render funcction which renders the grid
    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

Part B

m_startPosChosen = bool to set a cell as the start cell in the search

M_endPosChosen = bool to set a cell as the end cell in the search

M_INFINITY= double which is set to the max value possible divided by 10

setNeighbours() = sets the neighbours of a cell

setPredecessors() = sets the predecessors of a cell

setupGrid() = sets up the grid

heuristic() = calculates the heuristic value of a cell

render() = used to render the grid

Table 2-2 Description of The Utilised Class Images

Algorithm’s classes: UI and games class images	Description

```
/// <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

returns the cell with the lower key value or in the case of an equal key value return the cell with lowest key value.

```
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

K_M = key modifier which is the offset for change in start position of the search

dstarLiteTimer = timer to track time for completion of the search

dstarGoalFound = termination condition of the search

```cpp
// 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

getTimer() = returns the time for search completion

getDstarPathFound() = returns the termination condition

U_pq= priority queue which holds the cells and compares them against eachother

DstarLiteMain()= main loop of dstar lite

updateVertex() = updates the cells during the search

ComputeShortestPath() = computes the shortest path

initDstar() = initialises the grid for dstar lite to work

calculateDstarKey() = calculates the key for each cell

s_Last = used for tracking position of robot

```cpp
/// <summary>
/// compares the fcost  of cell 1 against cell 2's f cost to return the lower of the two
/// 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”.

It compares each cell based on their hcost + their gcost and returns the lower of the two.

```cpp
// the astar class
class Astar
{
	// private variables and methods

	// 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 m_AstarDone = false;

	// public methods and variables
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);

	// the stack to store the path astar has found
	std::stack<Cell*> m_stack;

	// contructor
	Astar();

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

“Astar” class as declared in the header file.

m_astarTimer= timer used to track time of search completion

m_AstarDone = termination completion

getTimer() = returns m_ astarTimer

getIdDone() = returns m_ AstarDone

AstarInit()=initialises the grid for astar to work

computeShortestPath() computes the shortest path using astar

m_stack = holds the path

Astar()= default constructor

~Astar() = default destructor

```cpp
/// <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”.

returns the cell with the lower key value or in the case of an equal key value return the cell with lowest key value.

```cpp
// life long planning a star class
class LpaStar
{
	// private variables and methods of the class
private:

	// 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 m_EPS = 2.0f;

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

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

	// temination condition
	bool m_LPApathFound = false;
```

Private members of the Lifelong Planning Astar Class

m_clock = clock used to track time for completion

m_K_M = key modifier works as offset for change in heuristic

m_LpaStarTimer = timer used to track time for completion

m_LpaPathFound= termination condition

```cpp
	// public methods of the class
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* t_node, Cell* Goal, Grid* t_grid);

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

	// default constructor
	LpaStar();

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

Public members of the Lifelong Planning Astar Class

Set/Get terminationCondition()= sets and gets the termination condition

getTimer() = returns m_LpaStarTimer

initLPAStar() = initialises the grid for lpa star to work

LPAStar() = computes the shortest path using the lpa star algorithm

updateNode()= updates each node in the path

CalulateKey() = calculates the key to a cell

LpaStar()=default constructor

~LpaStar() = default destructor

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

Functor used in "Dijkstra's" search algorithm.

It compares each cell based on their gcost and returns the lower of the two.

```
// the dijkstras search class
class Dijkstras
{
    // private class variables
private:

    // boolean to check if the goal has been found termination condition
    bool m_djkstrasPathFound = false;

    // timer to track the time taken of the search
    sf::Time m_dijkstrasTimer;

    // public variables of the class
public:

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

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

    // computes the path for dijkstras search algorithm
    void computeShortestPath(Cell* t_start, Cell* t_Goal, Grid* t_grid);

    // default constructor
    Dijkstras();

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

"Dijkstra's" Class as declared in the header file.

m_dijkstrasPathFound = termination condition

m_dijkstasTimer= timer used to track time for completion

getTimer() = returns m_dijkstrasTimer

getDijkstrasPathFound() returns m_dijkstrasPathFound.

computeShortestPath() =computes the path using dijkstras search algorithm

Dijkstras() = default constructor

~Dijkstras()= default destructor

```cpp
// the depth first search class
class DepthFirstSearch
{
	// private class variables
private:

	// the clock used to track the time
	sf::Clock m_clock;

	// boolean to check if the goal has been found to terminate the search
	bool m_depthGoalFound = false;

	//timer to track the search time
	sf::Time m_depthfirstSearchTimer;

	// stops the timer reseting and any bools etc from doing so
	bool initComplete=false;

	// public class variables and methods
public:

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

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

	// sets the timer bool
	void setTimerBool(bool t_b);

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

	void initDepth(Cell* t_curr, Cell* t_goal, Grid* t_grid);

	// default constructor
	DepthFirstSearch();

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

"Depth First Search" Class as declared in the header file.

m_clock = clock for timer

m_depthGoalFound = termination condition

m_depthfirstSearchTimer = timer used to track timer for completion

initComplete = bool for initialisation

getTimer() = returns m_depthfirstSearchTimer

setTimerBool() = used to reset timer

computeShortestPath() = computes the path using depth first search algorithm

initDepth() = initialises the grid for depth first search

DepthFirstSearch() = default constructor

~DepthFirstSearch() default destructor

```cpp
// menu class
class Menu
{
	// pivate member variables
private:
	// objects used inside of the menu class
	GridSize m_gridSwitcher;
	WhichAlgorithm m_slgSwitcher;
	Race m_raceDecider=Race::No;
	debug m_debugDecider=debug::Off;

	// sf::rectangle shape
	sf::RectangleShape m_rect;

	// vector which holds all of the rect shapes in the class
	std::vector<sf::RectangleShape> m_rectVec;

	// font used for the text
	sf::Font m_font;

	// all of the text objects used in the class
	sf::Text m_text[13];

	// intial y and xposition of the shapes and text
	float m_yPosition = 40;
	float m_XPosition = 10;

	// offset of each shapes position
	float m_offset = 300;
```

“Menu” class as private member variables in the header file.

Part A

m_gridSwitcher = controls grisSize Enum

m_slgSwitcher = controls WhichAlgorithm Enum

m_raceDecider = controls Race Enum

m_debugDecider = controls debug enum

m_rect = rectangleShape

m_rectVec = holds all the sf::rectangleShapes

m_font = font used

m_text[] = array which holds all of the text

m_yPosition = initial y position of the rectangleShape

m_xPosition = initial x position of the rectangleShape

m_offset = offset for positions

```cpp
// positional values for the rectangle shapes and text
float m_leftColXpos = 10;
float m_middleColXpos = 310;
float m_rightColXpos = 610;
float m_topRowYpos = 210;
float m_middleRowYpos = 380;
float m_bottomRowYpos = 590;
float m_veryBottomYpos = 700;
float m_positionOffset = 150;

// size offset
float m_XsizeOffset = 190;
float originalSize = 150;

// number of text and rectangle shapes to be created
const int m_MAX_TXT_RECTANGLES = 13;
```

“Menu” class as private member variables in the header file.

Part B

From m_leftColXpos to m_positionOffset are all positional values and offsets

m_XsizeOffset = used for change in rectangleShape size

originalSize= original value for size

m_MAX_TXT_RECTANGELS = max text allowed

```cpp
    // publuc methods used inside of the class
public:

    //default constructor
    Menu();

    // default destructor
    ~Menu();

    // returns which algorithm that has been chosen for use inside of the menu
    WhichAlgorithm& getalg();

    // reuturns if the algorithms are going to race or not
    Race& getRaceStatus();

    // returns if you can see the variable data for dstar lite
    debug& getdebugStatus();

    // returns the rectangle shapes used to make the menu
    std::vector<sf::RectangleShape> getVec();

    // returns the grid size chosen for use
    GridSize& setGridSize(sf::RenderWindow & t_windowTwo, Grid & t_grid, Grid& t_gridTwo, Cell *t_cell);

    // render function of the class
    void render(sf::RenderWindow& t_window);

};
```

“Menu” class as public member variables in the header file.

Menu() = default constructor

~Menu() = default destructor

getalg() = returns m_slgSwitcher

getRaceStatus() = returns m_raceDecider

getdebugStatus() = returns m_debugDecider

getVec() = returns m_rectVec

setGridSize() = sets and gets the grid size also selects the algorithm to be used

render()= renders the menu

```cpp
class Game
{
    // public variables and methods of the class
public:

    // default constuctor
    Game();

    // default destructor
    ~Game();

    // objects used inside of the class
        // which algorithm is being used controller
    WhichAlgorithm m_switcher;
        // which grid size is being used controller
    GridSize m_gridSizeState;
        // if the algorithms are to be raced controller
    Race m_raceState=Race::No;
        // if the debug visual is on or off controller
    debug m_debugState=debug::Off;
        // if the app is in play or test mode controller
    Mode m_mode = Mode::PLAY;
        // astar object
    Astar m_astar;
        // Dijkstras object
    Dijkstras m_dijkstras;
        // DepthFirstSearch object
    DepthFirstSearch m_depthFirstSearch;
```

Public members to class "Game", as declared in the header file.

Part A

Game() = default constructor

~Game() = default destructor

m_switcher= switched algorithm used

m_gridSizeState = switchers grid size

m_raceState = controls if algorithms race

m_debugState= controls the debug rendering

m_mode = controls which mode the application is in

m_astar= Astar object used to run Astar search

m_dijkstras= Dijkstra's object used to run Dijkstra's search

m_depthFirstSearch = as depth First Search object used to run depth First Search

```
    // DstarLite object
DstarLite m_dStarLite;
    // LpaStar object
LpaStar m_LpaStar;
    // Menu object
Menu m_menu;
    // Grid object for first screen
Grid m_grid;
    // Grid object for second screen
Grid m_gridTwo;
    // JumpPointSearch object
JumpPointSearch m_jps;
    // IdaStar object
IdaStar m_ida;


void run();
// controls wall placement
bool m_temp = false;
```

Public members to class "Game", as declared in the header file.

Part B

m_dstarLite= Dstar Lite object used to run Dstar Lite search

m_LpaStar= Lifelong Planning A star object used to run Lpa star

m_grid = grid object for editable grid

m_gridTwo = used for grid with Dstar Lite

m_jps = Jump Point Search star object used to run jump point search

m_ida = Iterative Deepening A Star used to run Ida star

run() = run function used in game loop

m_temp = controls the wall placement

```cpp
	// private member variables of the class
private:

	// if game is in play mode you can place walls and star and end position
	void PlayMode();
	// all start and end points are random including wall placement
	void TestingMode();

	// booleans to set start and endpoints of a path
	bool m_SrtChosen = false;
	bool m_EndChosen = false;
	bool temp = false;
	bool tempOne = false;

	bool m_exitGame;

	// cell pointers for path construction
	Cell* m_tempsEnd;
	Cell* m_tempstart;
	Cell* m_tempstartTwo;
	Cell* m_tempsEndTwo;
	Cell *m_cellVAR;

	// stack to draw the path
	std::stack<Cell*> m_AstarStack;
```

Private members to class "Game", as declared in the header file.

Part A

PlayMode() = runs the application in play mode

TestingMode() = runs the application in testing mode

m_srtChosen = bool to check if a start cell has been chosen

m_EndChosen = bool to check if the end cell has been chosen

Temp = wall control on grid one

tempOne = wall control on grid two

m_exitGame = bool to control exiting the application

from m_tempsEnd to m_cellVar are all used to run the algorithms and find the star positions and end positions as cells

```cpp
    // the different windows used in the application
    sf::RenderWindow m_window;
    sf::RenderWindow m_windowTwo;
    sf::RenderWindow m_windowAstar;

    // for inputing data into the excell files
    ofstream m_outputData;


    // tracks the start and end cells chosen for both grids
    int m_startCelI_Id;
    int m_startCellTwo_Id;
    int m_EndCell_Id;
    int m_EndCellTwo_Id;

    // game loop functions
    void processEvents();
    void processKeys(sf::Event t_event);
    void processMouseInput(sf::Event t_event);
    void update(sf::Time t_deltaTime);
    void render();

};
```

Private members to class “Game”, as declared in the header file.

Part B

From m_window to m_windowAstar = all of the windows used in the application

From m_startCell_id to m_EndCellTwo_id= used to track the ids of each start and end cell on both windows

From processEvents() to render() = functions used in the game loop

Table 2-3 Description of The Code

2.3 Data structures used

The different types of data structures used in the projects are as follows:

1. Std::vector<Cell*> - generally used in the application for the building of the 2D grid
2. Std::vector<Std::vector<Cell*>> - is used to store the entire grid
3. Std::stack<Cell*> - this is used to store the paths in the application
4. Std::list<Cell*> - this is used to store the neighbours and predecessors of a cell
5. Static array of type text and int – used for text and rectangles shapes
6. Static Struct – used for the screen sizes
7. Object Class – used for algorithms menu, game and grid
8. Static Enum Class – used for different functions in the application
9. Std::queue<Cell*> - used during the searches
10. std::priority_queue<Cell*, std::vector<Cell*>, functor used > - used during the searches for different algorithm

2.4 Storing of Data

2.4.1 How is the Data Stored?

The Data is stored inside of an excel file for each algorithm. It sores the time which it takes for the algorithm to complete the path in seconds. Each algorithm has three separate excel files for the times stored on the three separate grid sizes from small, medium, and large grid sizes and as such the times stored reflect this.

2.4.2 When is the Data Stored?

The Data is stored after the algorithm has been run and the user can also select the testing mode which will give the algorithms a random start and end position. This will avoid any positional or path length bias as the path is completely randomised on the grid.

The Data was then collected and used for comparison purposes.

Code/File/Data Storing Images	Description
2 0.141228 3 0.134918 4 0.136207 5 0.138869 6 0.135377 7 0.137532 8 0.138712 9 0.137757	Example for the data stored in the excel file. The example is the data stored in “AstarTime.csv” this is the data collected on the small grid running the Astar Algorithm
AstarTime.csv AstarTimeLarge.csv AstarTimeMedium.csv	Example of the name of the algorithms excel file

if (m_gridSizeState == GridSize::small)
{
std::string AstarResult = std::to_string((m_astar.getTimer().asSeconds()));
std::ofstream outputData("AstarTime.csv", std::ios::app);
outputData << AstarResult << std::endl;
outputData.close();
}
	Example of how the algorithm times is stored in code.
	
	Description of code: This code collects the time taken for the algorithm to finish then inputs that time into the excel sheet.

Table 2-4 Storing of The Data

3 User Flow

1. The user opens the application

2. The user selects the grid size they want to run their algorithms on

3. The user selects the algorithm that they want to use

4. The user selects the start and end points of the search on the graph (start point selcted with mouse left click and end point selected with mouse right click)

5. The user will then see the path the algorithm takes

6. The user can place down walls on the grid and see how the algorithm reacts Note. This can be done before step 4 (wall placed will scroll wheel on the mouse)

7. The user will then see how the algorithm reacts to changes in the path

8. The user will then select to race the algorithm and will see the path Dstar Lite used

9. The user will then select to run debug and run a new path

10. The user will then see how Dstar Lite makes changes to the grid

11. The user will then select a different algorithm to see how it compares to Dstar Lite

12. Repeat step 11 until the user is finished with the application

13. The user can then see the times taken for each algorithm in their own excel sheets

4 Class Diagram

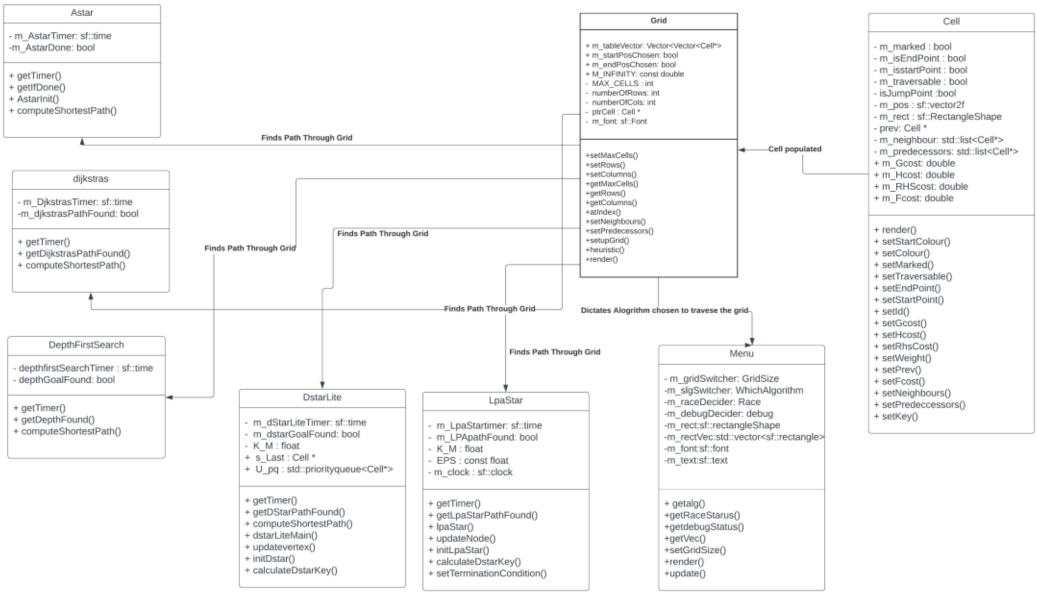

Figure 4-1 Class Diagram

5 CRC Cards

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

Figure 5-1 CRC Card "Grid"

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

Figure 5-2 CRC Card “Game"

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

Figure 5-3 CRC Card "Menu"

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

Figure 5-4 CRC Card "Astar"

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

Figure 5-5 CRC Card "Dstar Lite"

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

Figure 5-6 CRC Card "Dijkstras"

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

Figure 5-7 CRC Card "Depth First Search"

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

Figure 5-8 CRC Card "Lpa Star"

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

Figure 5-9 CRC Card "Cell"

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

Figure 5-10 CRC Card "Screen Size"

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

Figure 5-11 CRC Card "Mode"

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

Figure 5-12 CRC Card "Debug"

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

Figure 5-13 CRC Card "Race"

GridSize	
• enum class which controls the size of the grid	

Figure 5-14 CRC Card "Grid Size"

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

Figure 5-15 CRC Card "Which Algorithm"

6 Sequence Diagram

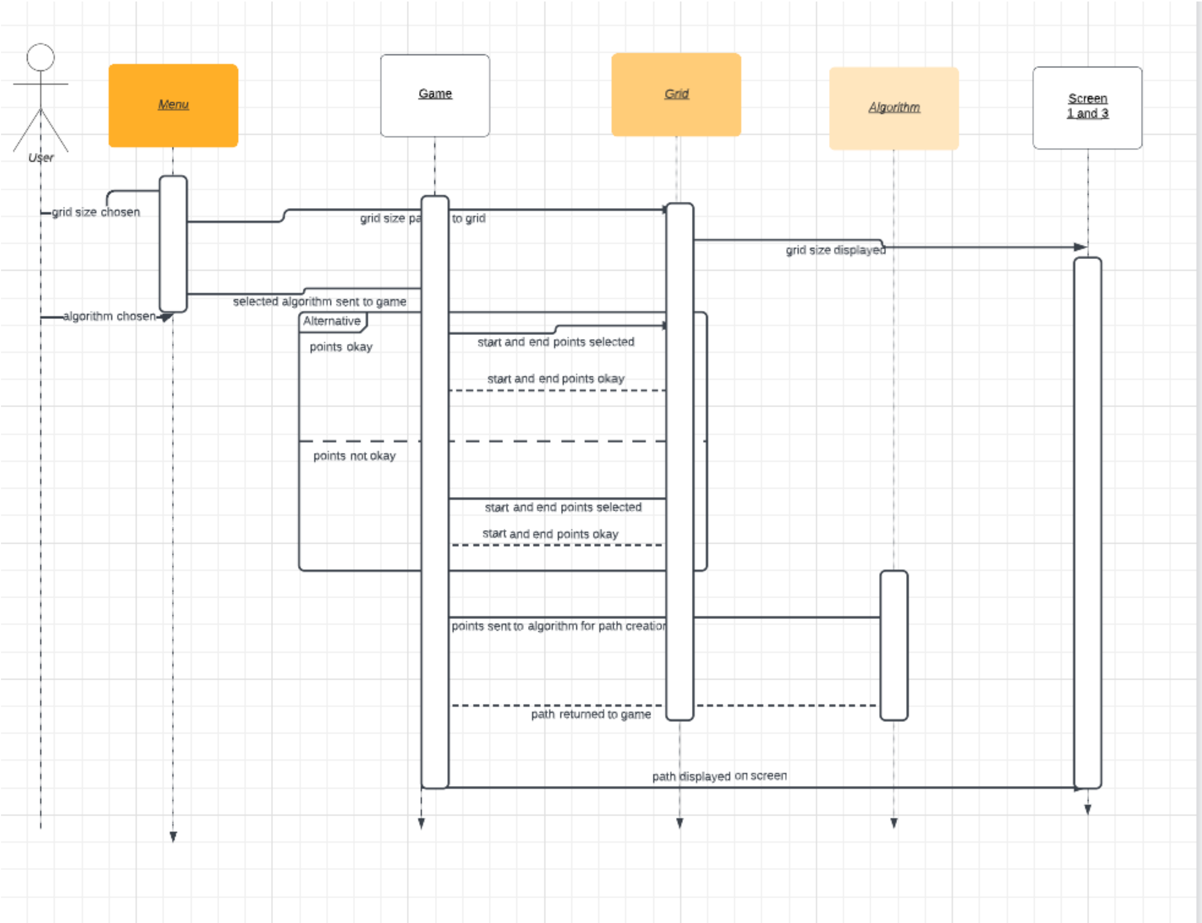

Figure 6-1 Sequence Diagram

7 Technologies

These are the technologies used for the completion of this application

Technology	Image
SFML 2.5.1	
C++	
Visual Studio 2022	
Excel	

Table 7-1Technologies used