Path Finding in a 2-D Grid with Obstacles

Daniel Butters

Issue: 1
Date: January 22, 2016

School of Physics and Astronomy
University of Birmingham
Birmingham, B15 2TT

Contents

1 Abstract

The aim of this project is to create a program that can plot a route between user defined start and end points in a 2 dimensional grid. This route should avoid obstacles that may prevent a direct path between the start and end points. This program should use object oriented programming (OOP) in C++ to implement classes needed for the pathfinding. This project uses OOP to implement a version of the A* PathFinding algorithm [2]. The program should take input from the user and display output graphically using the Fast Light Toolkit (FLTK) [1].

2 Introduction

The subject of the project and its underlying rules/laws

This project requires a path to be found between a start and end point while avoiding obstacles in a 2 dimensional grid. The rules of the path plotting are:

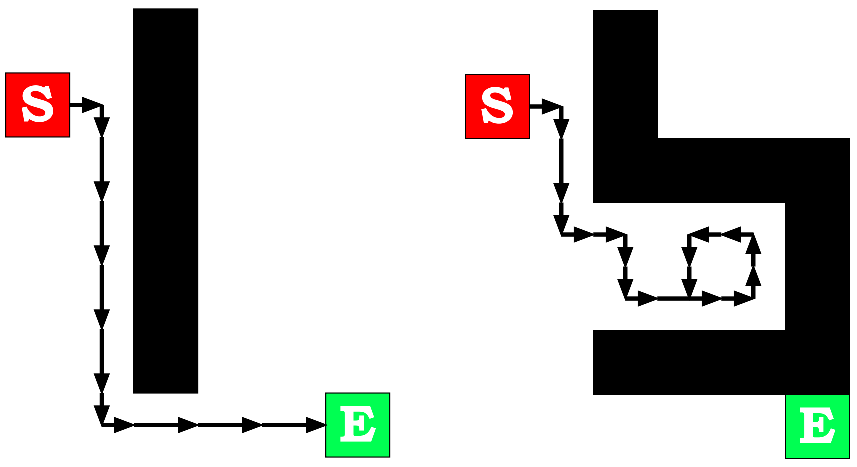
1. The route-plotting algorithm can travel in any direction, so it can travel horizontally, vertically or diagonally through the grid.
2. The route-plotting algorithm can only travel a distance of one square per iteration of the algorithm.
3. The route cannot include any squares which are obstacles.
4. If a path can be found, it must be found.

This report will explain how the problem was modelled, detailing the design that was created and the technical details of the solution. Three pathfinding methods are evaluated and the most suitable method is chosen. A short user guide will provide instructions for how to use the program to plot routes while avoiding obstacles.

3 Pathfinding Methods

3.1 Wall Following Method

There are multiple ways to find a route between two points. One could treat the obstacles as 'walls' in a building. In this system, if a 'wall' is encountered, the algorithm would turn to travel parallel to the wall, continuing onwards until it can turn back towards the end point. This would act in a similar way to a robot such as an automatic vacuuming device, that turns when it meets an obstacle. This model has a problem in that it is very much up to chance whether the robot will meet the goal as there is not guarantee that it will not get stuck in a recess in the 'wall'.



(a) Wall-Method with no recesses in the walls

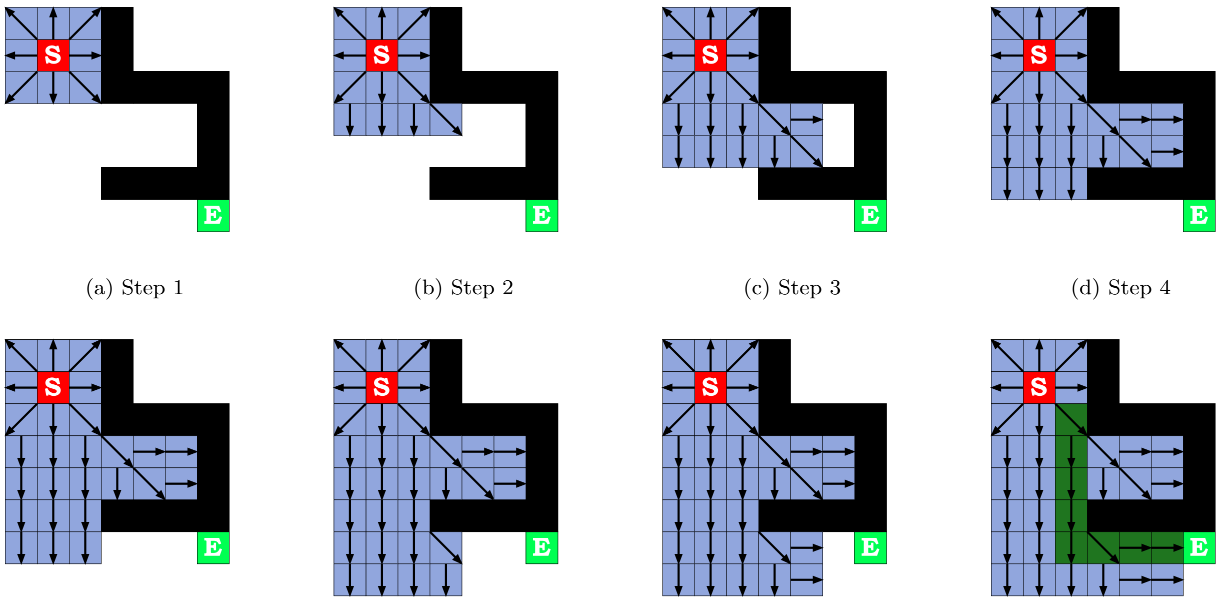
(b) Wall-Method with a recess in the wall

Figure 1: This shows how a method that changes direction when it encounters a wall shows how a recess in the wall could cause the algorithm to get stuck in a loop.

Figure 1 shows that when the obstacle is a simple wall that can be avoided by following parallel to the wall, the wall method is successful in finding a path. However, when there is a recess in the wall, the method would turn towards that recess. When it encounters a corner, it would turn away from that corner. This would cause the algorithm to turn around in the recess, then turn back towards the end point, forming a loop that can be seen in the second figure. This method essentially relies on random chance to find the route, so cannot be relied upon to always find the path to the end point in a reasonable time.

3.2 Stepping Outwards from the Start Method

Another way to find a route would be to incrementally step outwards from the start point in a square that increases in radius by 2 squares after each step. Obstacles cause the square to not progress further out from the start in that direction. The algorithm would continue to scan larger portions of the grid until the end point is found. This method is guaranteed to find a route to the end point eventually if one exists. However, it requires a large number of nodes to be considered for a large grid before the end point is found. Figure 2 shows that this method requires scanning a large number of nodes for even a small grid of 8*8 nodes. For a larger grid, the number of nodes requires would be much larger.

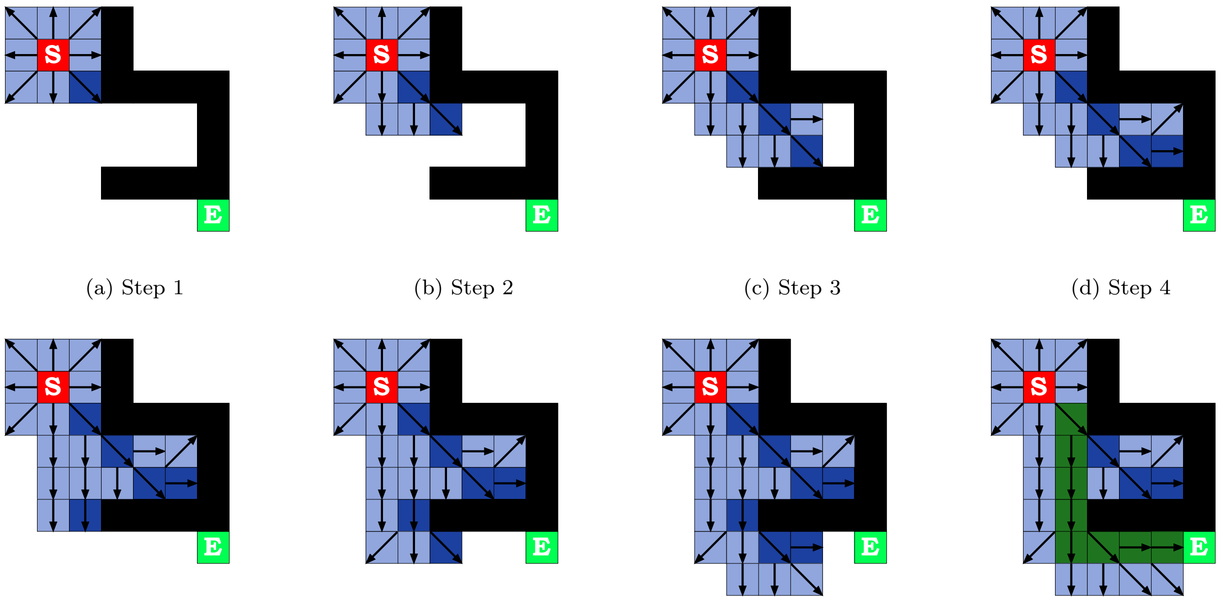


(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

(e) Step 5 (f) Step 6 (g) Step 7 (h) Step 8

Figure 2: This shows how a method that incrementally steps outward from the start point would build a route to the end point

3.3 A* PathFinding Method



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

(e) Step 5 (f) Step 6 (g) Step 7 (h) Step 8

Figure 3: This shows how a method that uses heuristics to step towards the goal would progress.

Another solution to this problem is to use a similar method to the stepping method described previously. However, this method would use heuristics to make an estimated guess for the best node to examine next, meaning that in general fewer nodes need to be examined before the route is found. This method is similar to how an autonomous system such as an animal decides which path to take to travel from point A to B. Typically an animal will travel in the direction of the end point, using estimations of distance to determine the best path to the goal, adjusting its route as it encounters new obstacles.

The A* pathfinding algorithm used in this project attempts to make similar decisions using heuristic estimations of the distance to the end goal. Figure 3 shows the process defined by the A* pathfinding method. This figure shows that for a given problem, the A* pathfinding method will generally require fewer nodes to be evaluated and so will be more computationally efficient than the stepping method.

4 Design

This project requires 4 classes: Position, Cell, Grid, and MapWindow

4.1 Position Class

This class is a 2-D vector used to store the positions of Cells in the grid. It has two attributes, fPosX and fPosY. This class is used when a position of a cell is needed to be set or returned. It is useful because it requires less manual manipulation of the x and y positions of a cell than if they were simply stored as two integers, and operators can be defined for Position that allow the program to add, compare etc. two positions without manually adding the x and y components. This class has a function Print() that prints the x and y components to the console in an easy to understand format using cout.

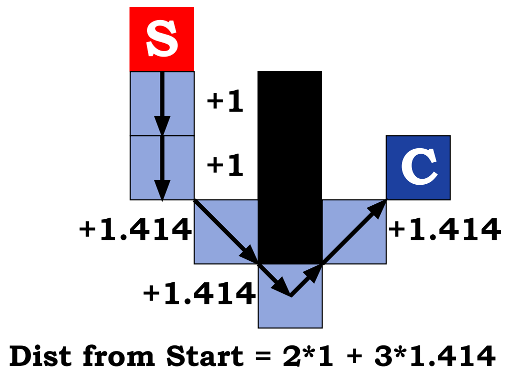
This class is mainly for convenience as an instance of the Cell class stores a position variable Position fPos which can be accessed using the function Cell.getPos().

4.2 Cell Class

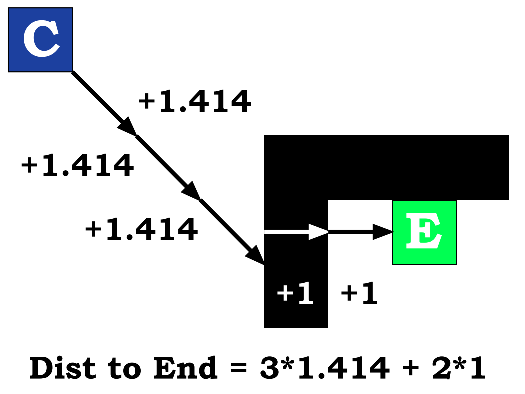
4.2.1 Attributes of a Cell

This class comprises the members of the vector of nodes stored in the Grid class. Each instance of this class has a Position, a Parent Cell, a Distance from Start, a Distance to End and a Type.

1. Position - this is the position of the node in the grid. It is a 2-D vector.
2. Parent - As each node is evaluated, it is assigned a parent node. This process builds up a chain of parents that will allow a route to be plotted.
3. Distance from Start - This attribute is derived as the node is assigned a parent. Initially, the Start cell has a Distance from Start value of 0. As cells assign the Start cell as a parent, their Distance from start is assigned as either 1, if the parent cell is horizontally or vertically adjacent to the child cell, or 1.414 if the parent cell is diagonally adjacent to the child cell. As a chain of parent cells builds up, the value of Distance from Start increases to express the length of the path from the Start cell to the current cell. Figure 4 shows how this variable is calculated.
4. Distance to End - this is an estimated distance from the current cell to the end cell. This is estimated by calculating the shortest path that exists between the current cell and the end cell assuming there are no obstacles between them. Figure 4 shows how this variable is calculated.
5. Type - This variable is a string that stores the type of the cell. This is useful for differentiating the cells. The cells can be one of 7 types: Start, End, Obstacle, Open, Closed, Path or UnTravelled. The purpose of these types is explained in the next section.



(a) How the distance from the Start cell to the current cell is calculated



(b) How the distance from the current cell to the End cell is estimated

Figure 4: These diagrams show how the variables Distance from Start and Distance to End are calculate. Notice how the Distance to End calculation ignores obstacles, so is only an estimate.

4.3 Grid Class

The Grid class contains a 2-D vector of Cells, cellArray, which makes up the 'Map' where the route is being plotted. This class is used to implement the algorithm which plots the route. This class also receives the user input from the MapWindow class, then sends the resulting output from the route plotting back to the MapWindow class to display the route to the user.

An instance of the Grid class is created using two integer inputs, fHeight and fWidth, that dictate the dimensions of the 2-D vector of cells, cellArray. cellArray is created and resized when an instance of the Grid class is created. The cells in cellArray can take one of seven types, each of which is defined in the next section.

4.3.1 Types of Cell

The Cells inside the cellArray vector can take 1 of 7 types:

1. Start - The starting point of the path. This cell is defined by the user and can be in any position in the Grid. This cell is chosen using button callbacks in the MapWindow class.

2. End - The end point of the path. This cell is defined by the user and can be in any position in the Grid. This cell is chosen using button callbacks in the MapWindow class.

3. Obstacle - The cells that cannot be traversed by the path plotted by the algorithm. These cells are defined by the user and can be placed in any point on the grid except the Start and End point. These cells are effectively obstacles because they cannot be added to the Open or Closed sets so they cannot be evaluated as a potential node in the path. These cells are chosen using button callbacks in the MapWindow class.

4. Open - These cells are members of the Open set. This means that they could potentially be evaluated to be part of the path. UnTravelled type cells are converted to Open cells when the Open cell with the lowest cost chosen by the algorithm is adjacent to the UnTravelled type cells.

5. Closed - These cells are members of the Closed set. The cell in the Open set with the lowest effective cost is chosen by the algorithm, and if it is not the End cell, the cell is converted from Open to Closed.

6. Path - These cells make up the path from the Start cell to the End cell. This is because they are part of the chain of parent cells that travels from the End cell to the Start cell.

7. UnTravelled - These cells are yet to have been added to either the Open or Closed Sets. They can be converted to Open cells when they are neighbouring an Open cell chosen by the algorithm with the lowest effective cost. This neighbouring Open cell will be assigned as their parent cell. When an instance of the Grid class is created, all cells are initially created as UnTravelled type.

4.3.2 Route Plotting using the Grid Class methods

The A* pathfinding method uses two sets of nodes:

1. The Open Set - this is a set of nodes that have yet to be evaluated by the algorithm. Nodes are added to this set by the algorithm when the 'current' cell has neighbours which are not obstacles or part of a set yet. This set is initialised with only the Start node as its member.
2. The Closed Set - this is a set of nodes that have already been evaluated by the algorithm and are not the end point. Nodes from the open set are removed from the open set and added to the closed set once they have been evaluated. This set is initialised with no members.

A node is evaluated using a sum of the two values Distance from Start and Distance to End. The sum of these values is the estimated overall 'cost' of using that node in the path from the start to the end point. During each iteration of the formula, the node in the Open Set with the lowest overall cost is the next node to be evaluated by the algorithm. When the next node is chosen, any valid nodes that are adjacent to the chosen node are added to the Open Set, so that they can be potentially be evaluated. This continues until the End node is found. The steps of the method are detailed below:

1. The Open Set is initialised with the Start node as its only member.
2. The Closed Set is initialised with no members.
3. The node in the Open Set with the lowest overall cost is chosen.
4. If the chosen node is the End node, display the path to the user.
5. If the chosen node is not the End node, add all neighbouring nodes that are not an Obstacle or in the Closed Set to the Open Set.
6. Set the parent of the neighbouring nodes as the current node.
7. Remove the current node from the Open Set and add it to the Closed Set.
8. Repeat Steps 3-7 until the current node is the End node or the Open Set is empty (there is no possible route if this happens).

This process is implemented using the PlotRoute function in the Grid class. It also uses other functions explained in the next section.

4.4 MapWindow Class

The MapWindow class creates an FLTK window which takes all input from the user and displays the output. An instance of this class is created after the instance of Grid class is created. The dimensions of the array of cells are passed to this class as well as a pointer to the instance of Grid class already created. This allows the MapWindow class to call methods in the Grid class and allows the button callbacks in the MapWindow class to change the cellArray in the Grid class.

This class has several components:

1. fMap - The pointer to the instance of Grid class already instantiated. This allows for transfer of information between the two classes.
2. Header - This is a box at the top of the window that displays instructions to the user. Button callbacks are used to change the label of this box so that each stage of the program is explained to the user.
3. Done Button - This button is disabled until the Start, End and Obstacle nodes have been defined by the user. This button calls the PlotRoute method in the Grid class. Once the results from the PlotRoute method have been displayed, the callback and label change to a function for closing the window and exiting the program.

4. Key - An array of buttons and boxes with labels that define which each button colour corresponds to each type.
5. buttonArray - A grid of buttons that correspond to each Cell in the Grid class. The button colours match the type of Cell at that position in the grid. These buttons are used for choosing the Start point, the End point and the obstacles. The callback functions send the position of the button to the Grid class so that the grid class can change the Cell at that position to the type designated. The Grid class can also change the colours of each button to match the output from the PlotRoute function.

5 How the Code Plots a Route

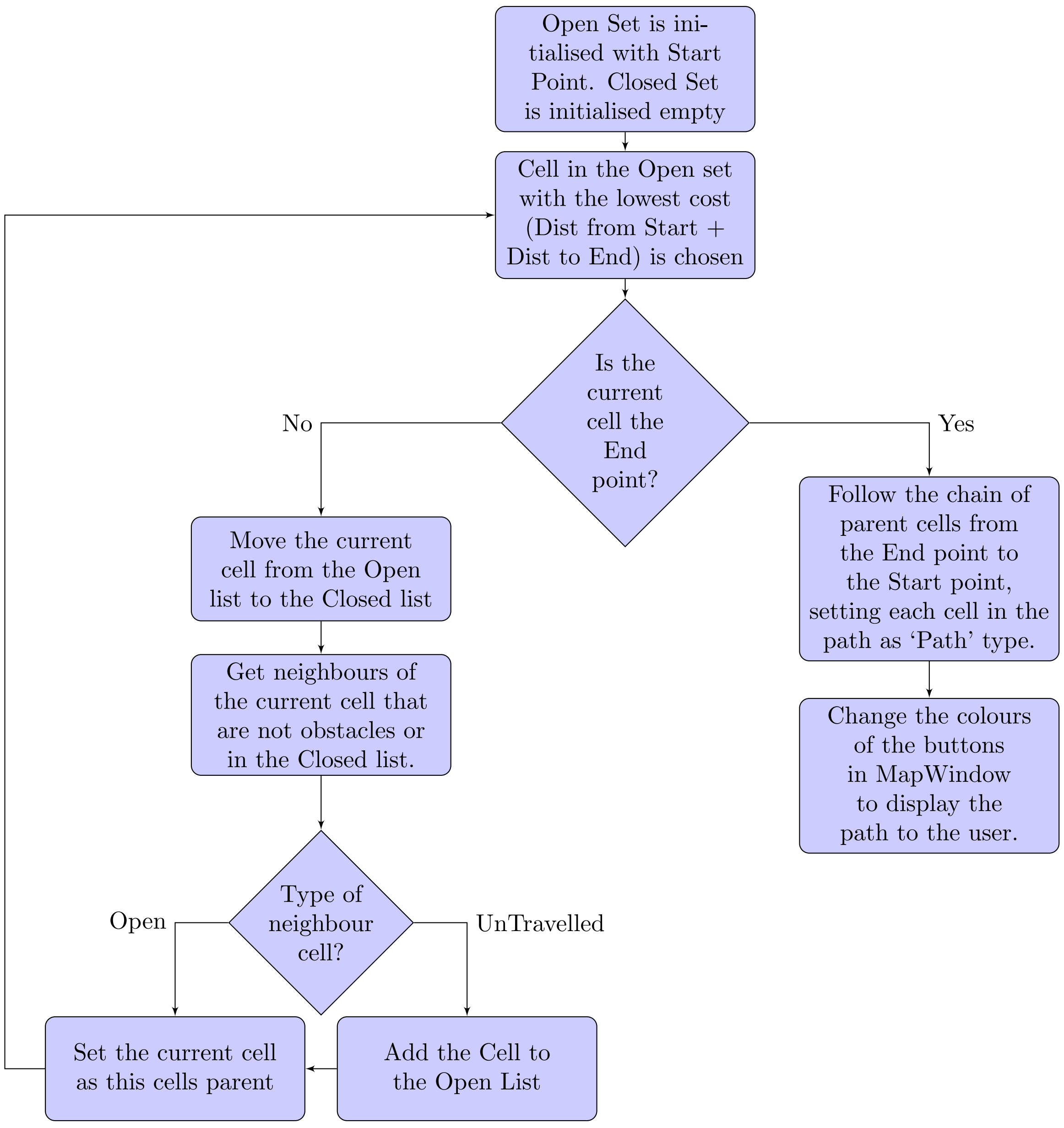

Open Set is initialised with Start Point. Closed Set is initialised empty
Cell in the Open set with the lowest cost (Dist from Start + Dist to End) is chosen
Is the current cell the End point?
No
Yes
Move the current cell from the Open list to the Closed list
Get neighbours of the current cell that are not obstacles or in the Closed list.
Type of neighbour cell?
Open
UnTravelled
Set the current cell as this cells parent
Add the Cell to the Open List
Follow the chain of parent cells from the End point to the Start point, setting each cell in the path as 'Path' type.
Change the colours of the buttons in MapWindow to display the path to the user.

Figure 5: This flowchart shows how the function PlotRoute finds a path between a given start and end point while avoiding obstacles.

Figure 6 shows the flow of the program from start to finish. Stages 4 and 5 are described in more detail in Figure 5, which shows the steps involved in the function 'PlotRoute' when it is called.

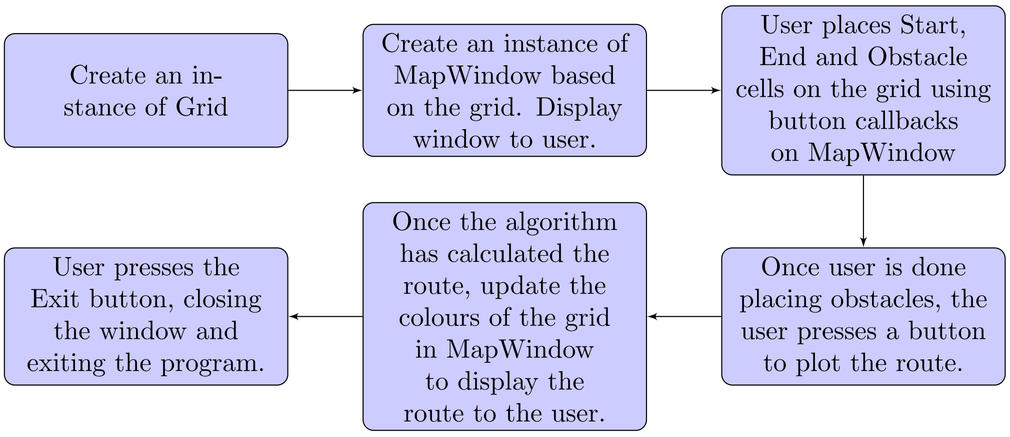

Create an instance of Grid
Create an instance of MapWindow based on the grid. Display window to user.
User places Start, End and Obstacle cells on the grid using button callbacks on MapWindow
Once user is done placing obstacles, the user presses a button to plot the route.
Once the algorithm has calculated the route, update the colours of the grid in MapWindow to display the route to the user.
User presses the Exit button, closing the window and exiting the program.

Figure 6: This flowchart shows the flow of the program from start to finish.

6 User Guide

The following figures detail the stages involved in plotting a route using this program. The start and end points can be assigned to anywhere in the grid. The Obstacles can also be assigned to anywhere in the grid, though it is advisable to make sure to include a possible route from start to finish when placing the obstacles. This is because if there is no possible route, the algorithm will spend a long time evaluating every square in the area surrounding the start point until it runs out of squares to evaluate.

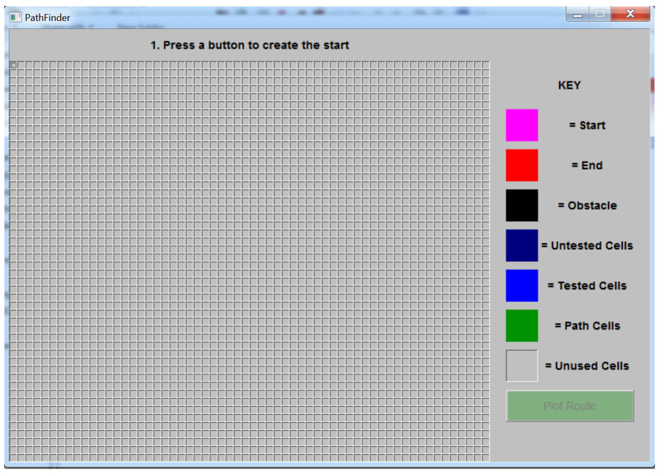


Figure 7: Stage 1 - The window opens with instructions to select a start point

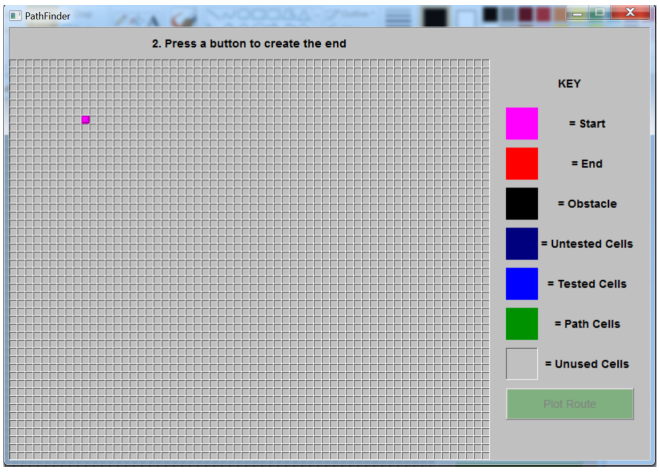


Figure 8: Stage 2 - The user clicks a button to select the start and is then instructed to select an end point

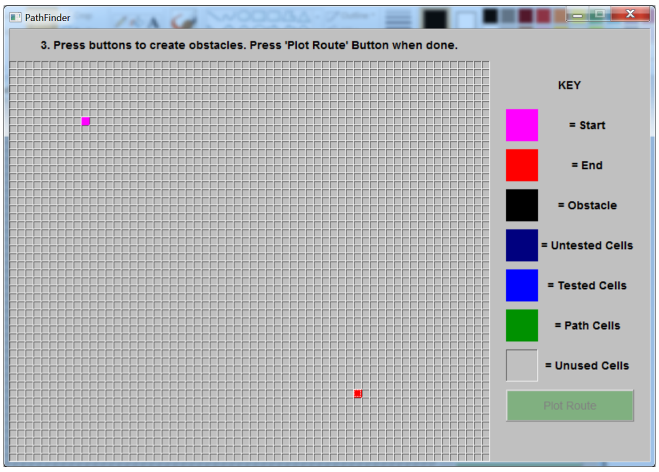


Figure 9: Stage 3 - The user selects the end point and is then instructed to place obstacles

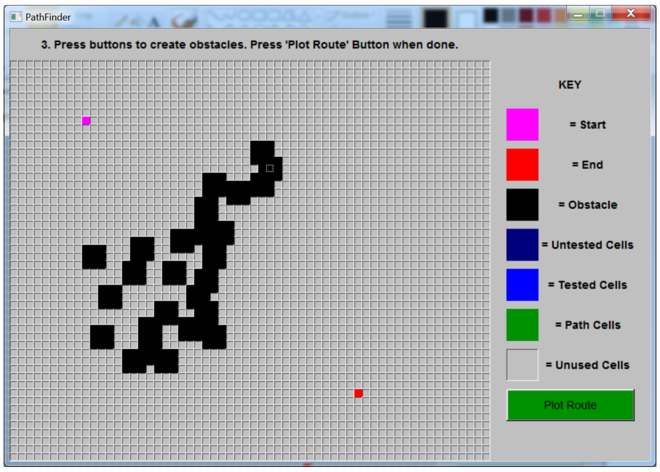


Figure 10: Stage 4 - The user places the obstacles. The ‘Plot Route’ button is then activated so that the user can press ‘Plot Route’ to plot the route

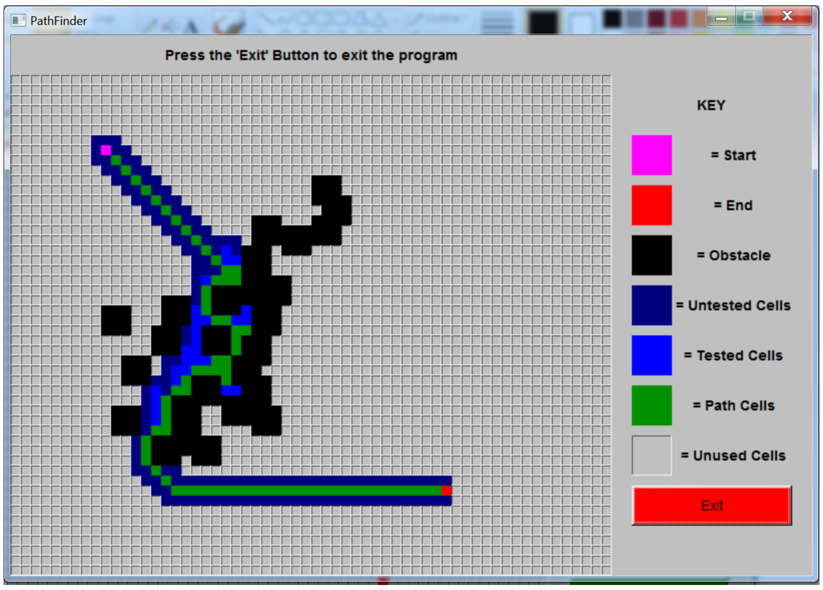


Figure 11: Stage 5 - The program plots the route then outputs the route to the screen, using buttons of different colours defined in the Key. The route plotted can be seen as the green squares on the grid. The user can then press 'Exit' to exit the program.

7 Conclusion

In conclusion, the aim of this project, to create an Object Oriented Programming based system to find routes between a user defined start and end point while avoiding obstacles has been achieved. The system, over numerous tests, has always found the route, no matter what obstacles have been placed in the way.

The main improvement on this system that I would make is to improve the route finding algorithm so that it would find the shortest path rather than just any path. This would be achievable by changing the criteria that decides which cells should be assigned which parent cells. There is a system that updates a cells parent if the route up to that point is shorter than the route which includes the previous parent cell. However, I could not get this system to work reliably and ultimately made the system liable to crash, so I decided to implement a more simple system which always found a route, but the route found is unlikely to be the shortest.

Another improvement I would make is to implement a system where a set array of obstacles could be placed rather than placing them manually. This could include such things as a maze, or a map which emulates the road system around a town so that the route finding could be shown in a setting more similar to how they are used in the real world.

References

[1] B. Spitzak and Others, FLTK Documentation, Updated 02/12/15, Accessed 20/01/16, `http://www.fltk.org/documentation.php`

[2] D. Imms, A* pathfinding algorithm, Updated 30/05/15, Accessed 20/01/16, `http://www.growingwiththeweb.com/2012/06/a-pathfinding-algorithm.html`