CS4006 - Intelligent Systems
A* Project Analysis

Seán Lynch (18245137)

Michele Cavaliere (18219365)

Nicole Berty (18246702)

Matt Lucey (18247083)

Table of Contents

Introduction

The shortest path problem involves finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimised[1]. In other words, given a vertex X and a vertex Y, we want to find the shortest path from X to Y to maximise efficiency. The shortest path problem can be defined for directed or undirected graphics, but in this case we are considering only the undirected graph definition on grid maps.

1 https://en.wikipedia.org/wiki/Shortest_path_problem

A* Description

The A-star algorithm is one of the best search algorithms to find the shortest path between nodes on a graph.[2] It is based on a breadth-first search algorithm, which means that it looks at all of the nodes at level 'n' before looking at the nodes at level 'n+1'. It uses heuristics to guide its search, and it is both optimal - meaning that is it guaranteed to find the best path (the one that costs the least) from the source to the destination - and complete - meaning that it will find all of the possible paths that can be formed from the start position to the end position. While A* has these benefits, it is a slow algorithm and it takes up a lot of space as it stores all of these paths. A* is summed up in this function f(n) = g(n) + h(n), where n is the next node on the path, g(n) = cost from start node to n and h(n) = estimated cost of cheapest path to goal node from node n. At each step of the path, A star chooses the minimum f(n) value for the next node n.[3]

Manhattan Distance

We use Manhattan distance as our distance heuristic function. Manhattan distance gets its name from the grid layout of Manhattan Island, New York. It is also known as the Taxicab metric.[4] Manhattan distance is the best heuristic function for the A* algorithm when traversing a grid in only 4 directions. It is calculated by d(i,j) = |xi - xj| + |yi - yj| where i and j are vectors on the grid and the x and y are their coordinates on a fixed cartesian plane.

Possible Applications of this Algorithm

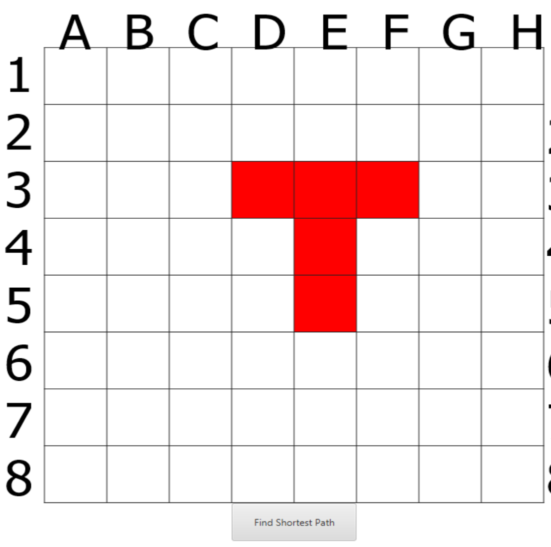
The A star algorithm can be used for maps, to find the shortest distance from the starting point to the destination. This can be used for real life maps or route planners, allowing us to find the shortest path between two cities, for example, or to take the shortest route to get somewhere. It can be used for video games in a similar manner - i.e. pathfinding. This allows the game's AI to plan their route in the game around complex obstacles.
This algorithm is also used in robotics, to allow robots to move around obstacles and get from one point to another in the quickest fashion.

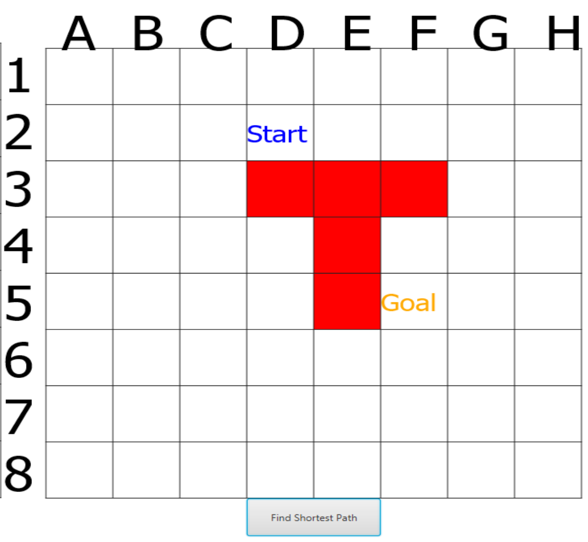
Implementation

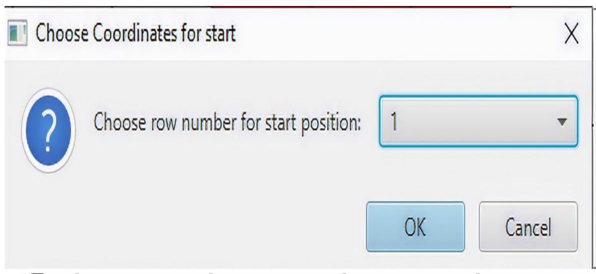
Explanation of the user interface (e.g., console or GUI)

We decided to use a GUI rather than a console interface and we used JavaFX to create this user interface. We started by making the 8 x 8 grid, which we did by creating both column and row constraints. We then labelled the top and left margin of the grid, the left with numbers (1 - 8) and the top with letters (A - H), representing the x and y coordinates of each square.

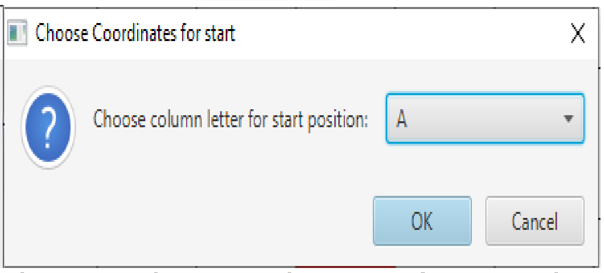
2 https://towardsdatascience.com/a-star-a-search-algorithm-eb495fb156bb
3 https://www.edureka.co/blog/a-search-algorithm/
4 https://en.wikipedia.org/wiki/Taxicab_geometry



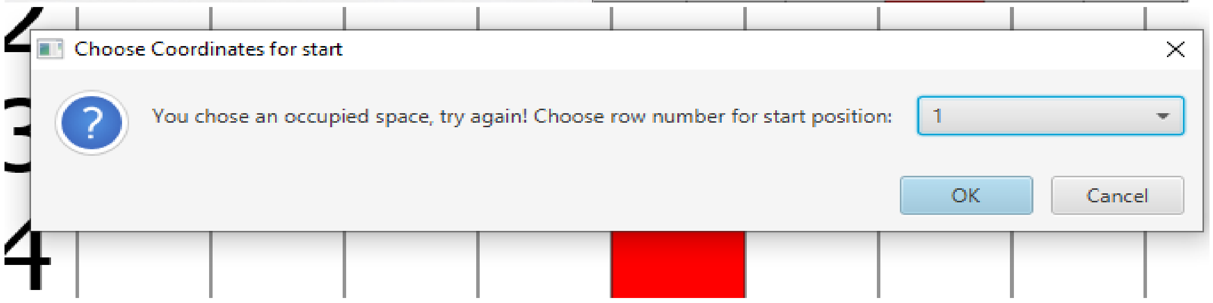



Choose Coordinates for start

Choose row number for start position: 1


Choose Coordinates for start

Choose column letter for start position: A


Choose Coordinates for start

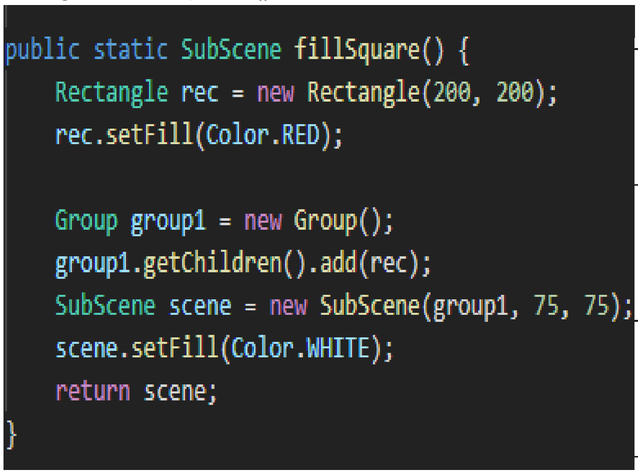
You chose an occupied space, try again! Choose row number for start position: 1

When the application is first opened, the graph is displayed in one window with the obstacle on it and four dialog boxes appear on the screen consecutively, allowing the user to input the x and y coordinates of the start and goal position. These dialog boxes contain a list of rows, 1-8, and columns, A-H, in a drop down list that the user must choose from - this means that it is not possible for the user to pick a start or goal position outside of the grid. However, it is possible for the user to pick a start or goal position that is on the obstacle, so we had to validate user input. If the start coordinates and/or goal coordinates are on the obstacle, the dialog boxes will reappear and the user will be asked to input a valid option.

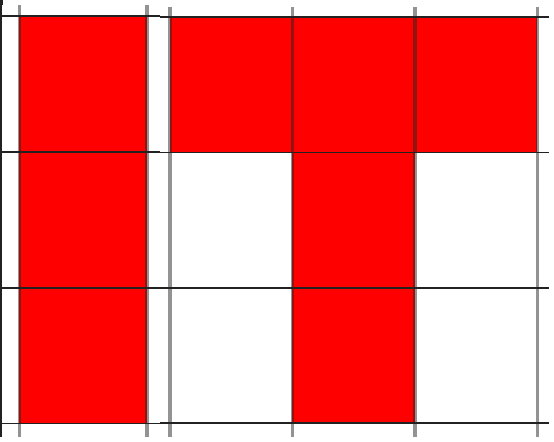
Once the user has chosen valid coordinates for start and goal positions, the grid is redisplayed with the start and goal nodes marked clearly, using the addText(Color col, String s) method. The “find shortest path” button prints the optimal path from the start position to the end position using the A* Algorithm, using an event handler attached to the button. All of these elements were added to a StackPane which was added to a scene and then to the primary stage in order to display the window.

Explanation of how you generate the obstacles

A random number is generated from 0-2, which determines the amount of squares added to the obstacle. Another random number is generated from 0-1, determining the orientation of said obstacle, where 0 = Vertical, 1 = Horizontal. Then the first node of the obstacle is randomly generated and the code builds the obstacle from there, whether it is an I, an L or a T. The obstacles are generated on the grid by filling in each node of the obstacle in red, using the fillSquare() method, as shown.



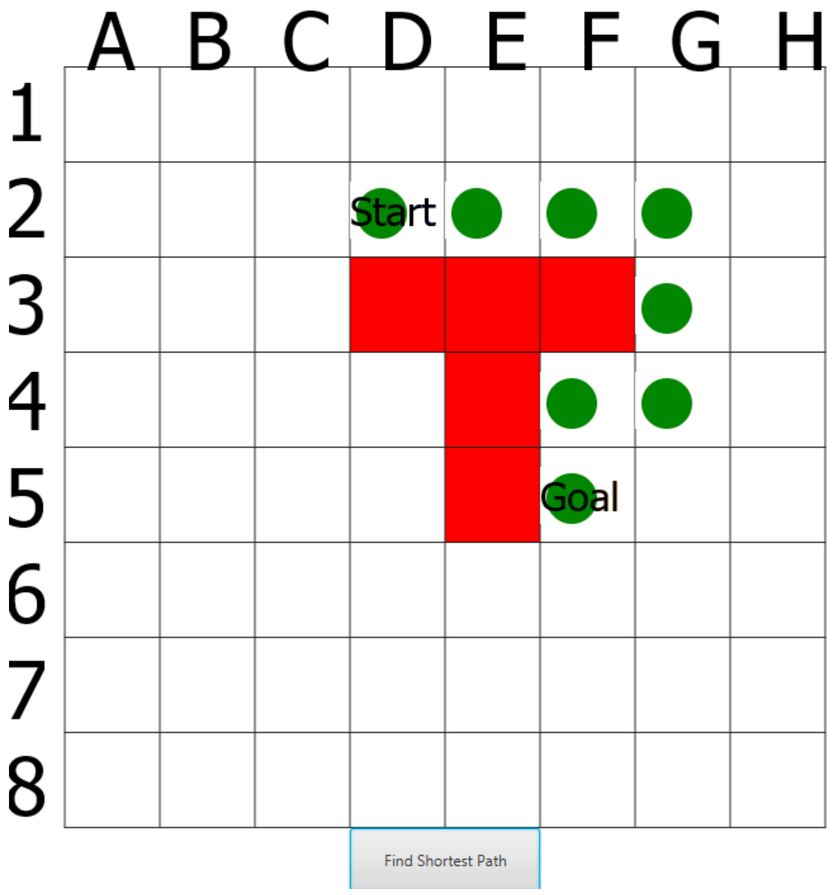
```
public static SubScene fillSquare() {
    Rectangle rec = new Rectangle(200, 200);
    rec.setFill(Color.RED);

    Group group1 = new Group();
    group1.getChildren().add(rec);
    SubScene scene = new SubScene(group1, 75, 75);
    scene.setFill(Color.WHITE);
    return scene;
}
```



Explanation of how to find the shortest path using screenshots of the user interface

Our A* implementation works as follows.

- We create two arraylists of type Node, Open and Closed.
- We add the start node to Open.
- We set the loop to continue until the current node equals the end node.
- If the Open list is ever empty the function will return null, as that would mean there is no path to the target.
- We use a for loop to choose the node with the lowest h value using the Manhattan distance.
- We remove the current node from the Open list and add it to the closed list.
- We find the neighbours of the current node and calculate their F, H, G values.
- We look through the neighbours of current and add them to Open if they aren't in either list.
- The while loop ends when the current node equals the end node.
- We then use a while loop to trace the path back from the end node to the start node
- The user presses the "Find Shortest Path" button on the GUI and the shortest path is printed.
- To print the path, we iterate through the arraylist which contains the shortest path and add a green circle to each square in the list using the pathDraw() method.



Conclusion

We use JavaFX to generate the GUI, display the board, its coordinates, and the shortest path, as well as the button to run the algorithm. We use dialog boxes to take user input for the start and goal position. The obstacle’s shape, orientation and position are generated randomly when the board class constructor is called to create the board. We implemented the A* algorithm using a variety of methods to calculate heuristics, like the Manhattan distance, as well as the values of h(n), g(n) and, subsequently, f(n) - which is calculated by the values(Node node) method in the Graph class.