Interactive, Visual Comparison of Single-Source, Single-Destination, Shortest Path Pathfinding Algorithms

Calvin James Newn

Analysis

Background

Pathfinding algorithms are a finite series of precise, well-ordered steps a person or computer can follow to determine the most efficient path from one point to another. Applications include: A GPS system to determine the fastest route from one location to another, path planning for a robot, or a computer game where an enemy A.I. is chasing the player.

However, these algorithms are often long and tedious to follow out manually. Thus, it is inefficient for a human to carry out an algorithm on his own. There is also a difference in speed in which the algorithms are carried out between different people, so we cannot easily make a fair comparison between algorithms when done by people. A computer-based solution to this problem is therefore optimal as a computer can easily compute multiple algorithms simultaneously and output results quickly.

There are many different pathfinding algorithms ways of implementing them which you can choose from, each with variable efficiency. However, an algorithm that is suitable for one situation may not be as suitable for another. Selecting the best algorithm for a particular task may therefore be quite difficult.

The Problem

The problem is being investigated is the comparison of different single-source, single-destination, shortest path algorithms operate on a finite two-dimensional grid. Obstacles may be placed between the start and destination. Movement between tiles is restricted to up, down, left, and right only.

The program will serve primarily as a visual pathfinding algorithm comparison tool. It also serves as a tile map editor.

I will be working with my computer science teacher, Simon Carter, who I have assigned as my project supervisor to produce a suitable solution to the problem.

Research

The most popular shortest path algorithms suitable for the problem seem to be the A*, Dijkstra, and D* algorithms. A* seems to work better than Dijkstra's whereas D* outperforms A* on finding the shortest path where the map data is constantly changing over time.

Pathfinding algorithms may have different heuristics - Different techniques of solving a problem more quickly than the traditional method. For example, the A* algorithm has Manhattan, Euclidean, and Chebyshev heuristics. Another variant is the Bi-Directional search, in which the search is done on both the source and destination nodes.

There seem to be close links between algorithms too. For example, the Breadth-First Search (BFS) works similarly to Dijkstra's Algorithm in unweighted graphs but is more efficient - Worst case performance for BFS is O(|V| + |E|) whereas Dijkstra's is O(|V|log|V| + |E|), where |V| is the number of nodes and |E| is the number of edges. The A* algorithm seems to be a combination of Dijkstra's algorithm and the Best-First Search algorithm where rather than searching for tiles away further away from the destination, it prioritises those that are closer to it.

There are also other algorithms, such as Depth-First Search and Best-First Search, which do not find the shortest path. Regardless, the Best-First Search algorithm may be included regardless to demonstrate how similarly it works to the A* algorithm.

The grid may be modelled through adjacency lists or adjacency matrices to allow for pathfinding. For this problem, an undirected, weighted graph would be needed

Difference between different types of graphs - In my case this will be a weighted undirected graph

Sources:
https://en.wikipedia.org/wiki/Pathfinding
http://theory.stanford.edu/~amitp/GameProgramming/

Potential users and their needs

The user of the program would be anyone who is interested in comparing different pathfinding algorithms by testing how each one performs on the same test map. An example could be a game developer who is interested in implementing an efficient pathfinding algorithm into his game, or an engineer who has difficulty choosing the best pathfinding algorithm for his robot navigation system.

To do this, the user will require a grid which they can add obstacles of their choice. Each obstacle must have a unique colour to differentiate between one another. The weights of each obstacle will represent how difficult it is to cross that particular tile; the greater the value, the harder it is. This value must also be editable.

There must also be different pathfinding algorithms to choose from. To compare these algorithms directly, copies of the map must be created. Each map must have a different pathfinding algorithm running. The steps taken and the time for each algorithm to finish must also be displayed to user once the algorithm has finished running.

To allow for different sized maps, the user should be able to resize the grid, preferably without losing current map data. Note that the entire map may not fit into the screen, so the user should be able to navigate through the map. To avoid confusion, ‘grid’ is defined as the visible part of the map on the screen, and ‘map’ is defined as the entire tile map, in which its size is defined by the user.

Not all users will need to work on weighted graphs, some may require a grid that only has an empty tile and a wall (Note: We do not have to give the wall a weight - All we need is to treat a wall as an ‘out of bounds’ tile). Therefore, there should be a toggle between unweighted and weighted graphs.

A user learning to implement a pathfinding algorithm should also be able to run through the algorithms step-by-step to obtain a better understanding of how the algorithm works. The working values of the algorithms should be outputted to each tile appropriately.

The user may also want to save and load map files so that they may transfer custom maps to other users or continue an investigation on a custom map at a later date, so this needs to be a feature in the program as well.

Limitations

The main goal of the program is to compare pathfinding algorithms. Although map creation is required for building a test map to do so, it is not the main focus of the program. There may not be enough time to create more convenient features such as being able to move a selection of blocks. These features may be added after the main program has been completed.

Objectives

Below is a set of objectives to check if the program has met the user's needs. Note that the list is a high-level overview of what should be accomplished by the end of the project.

1. The program should display a 2D grid of tiles with a valid 'size' (Number of tiles in a row and column) defined by the user.

2. The user should be able to create their own obstacles with custom name, weight, and colour.

3. The user should be able to add obstacles of their choice onto the grid.

4. The user should be able to clear the map.

5. The user should be able to navigate through the map if it does not all fit into the screen.

6. The user should be able to choose between at least 3 different pathfinding algorithms.

7. For all pathfinding algorithms, the shortest path from the starting tile to the destination tile should be displayed after it has been found.

8. The user should be able to compare multiple pathfinding algorithms of their choice by displaying copies of the map, each with a different pathfinding algorithm running. By pressing 'play', all chosen pathfinding algorithms should start simultaneously.

9. The steps taken and time taken for a pathfinding algorithm should be displayed after it has finished executing.

10. The program should have a 'step-by-step' mode, for all instances of the map, where one step of each pathfinding algorithm is done per button press.

11. The user should be able to resize the grid. If resizing up, then no map data should be lost. If resizing down, then inform the user that data may be lost.

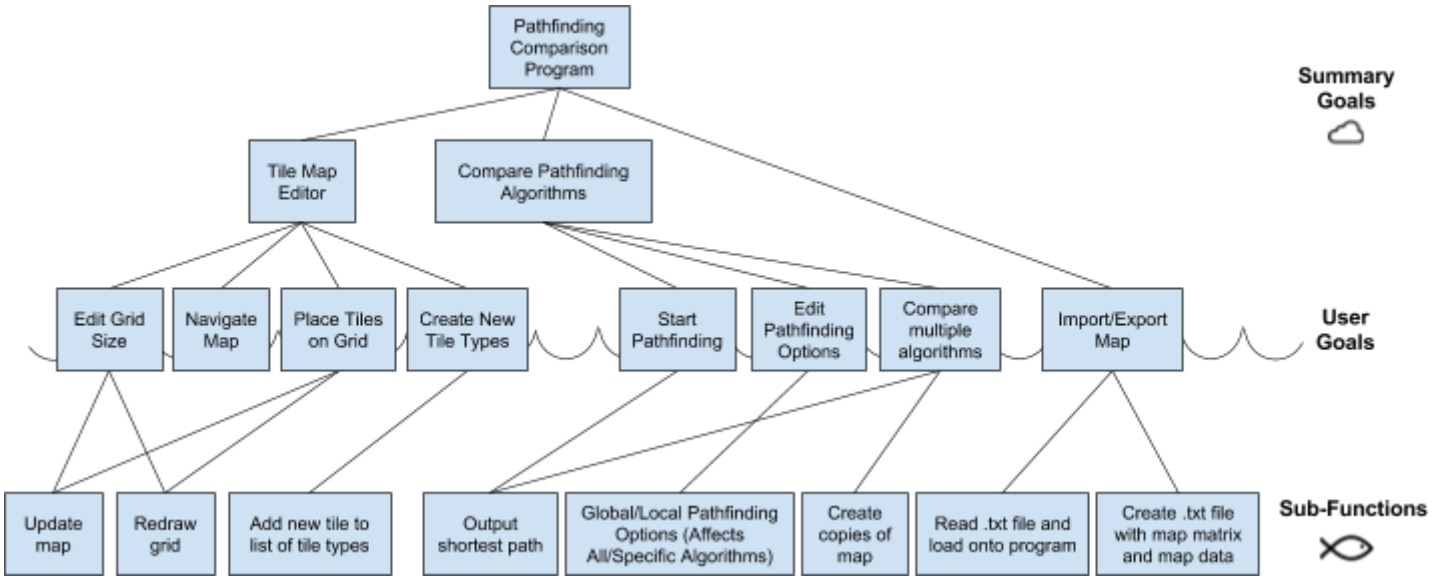
12. The user should be able to switch between 'weighted mode' and 'unweighted mode'. In unweighted mode, all current non-empty and non-wall tiles should be treated as a wall, but not deleted off the map.

13. The user should be able to save the current map file into a text file. The program should also be able to load custom maps.

Modelling the Problem

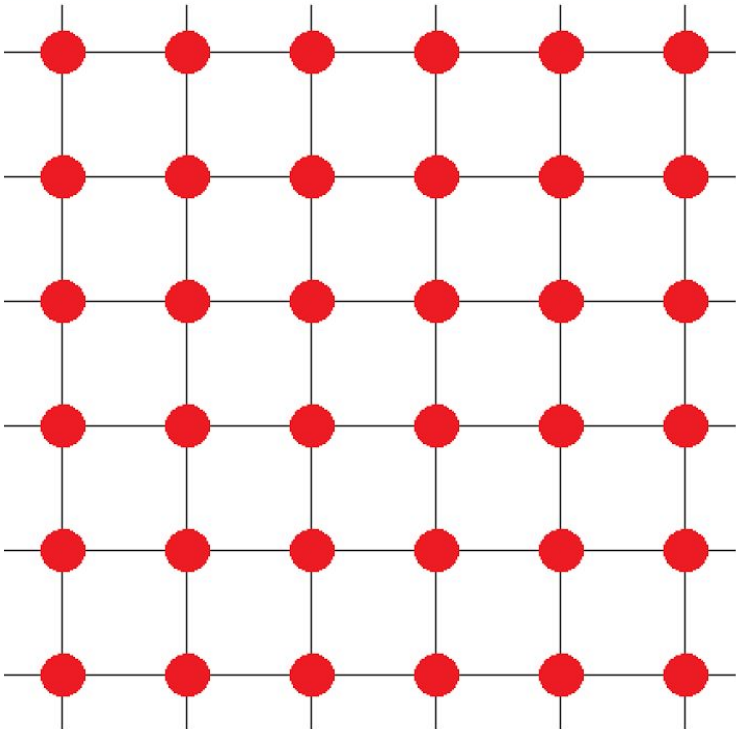
Use Case

Below is a ‘Use Case’ hierarchy diagram of the program. Goals are divided into goal levels - Higher-level goals are on top and lower-level goals are on the bottom. The ‘User Goals’ represents the user’s interaction with the program.

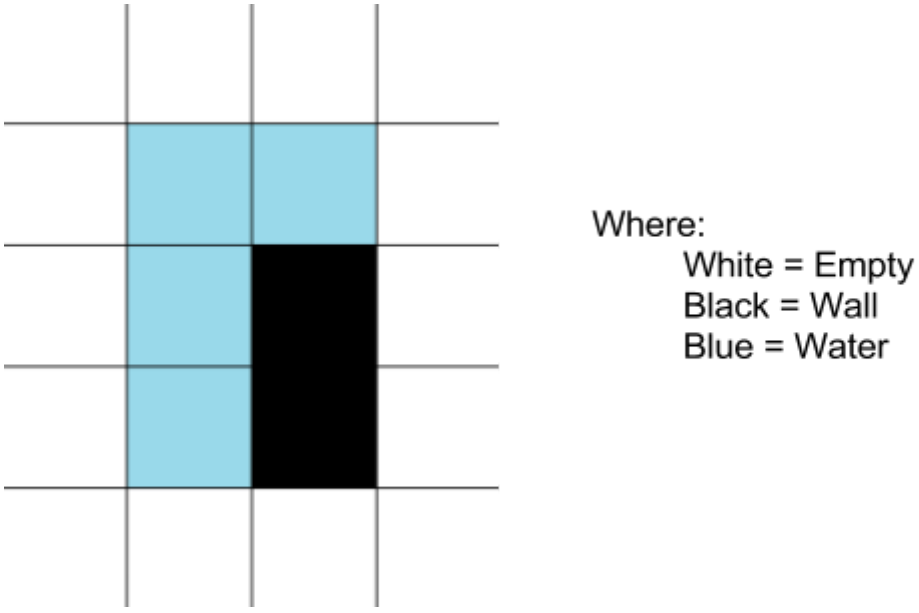


Grid Design

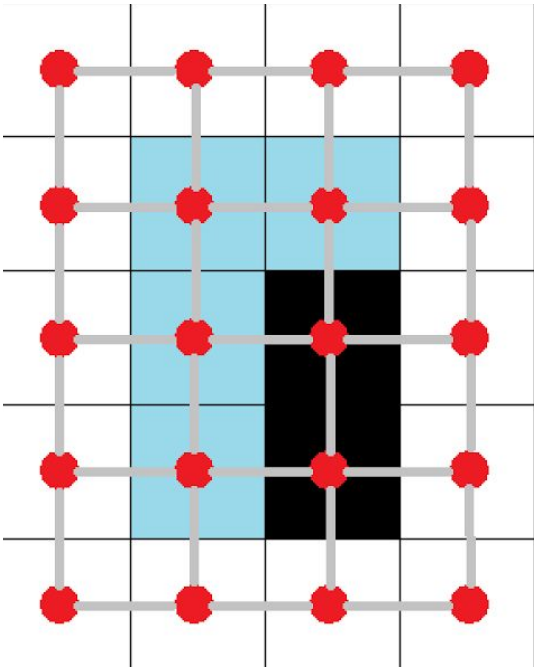
The tile map can be modelled as a graph, where each tile is a node and movement between nodes is restricted to up, down, left, and right only. In the diagram below, the red dots represent nodes and lines connecting these nodes represent edges:



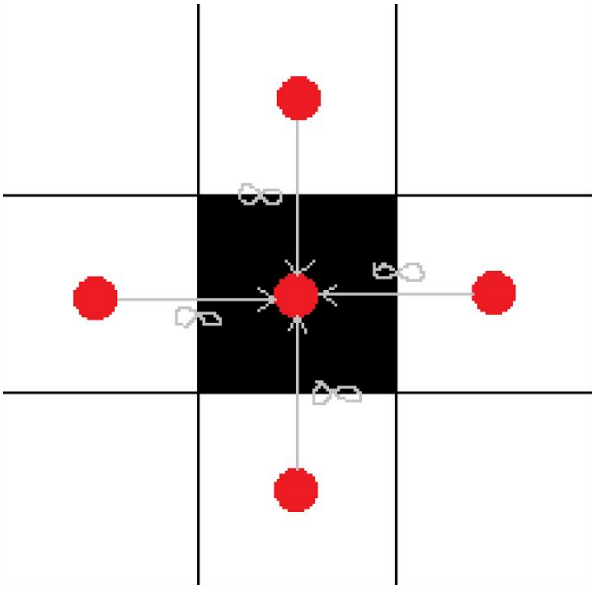
For an unweighted graph, all edges have the same weight. However, complications arise when we try to implement a weighted graph. Take the following grid as an example:



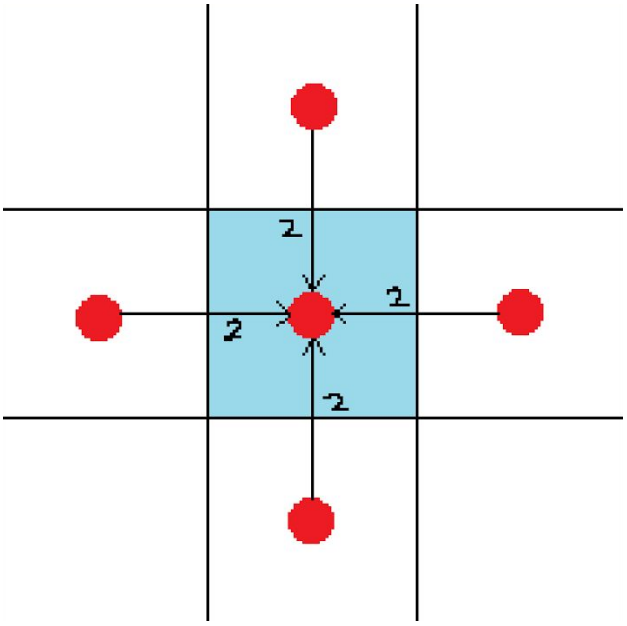
Modelling this as a graph produces:



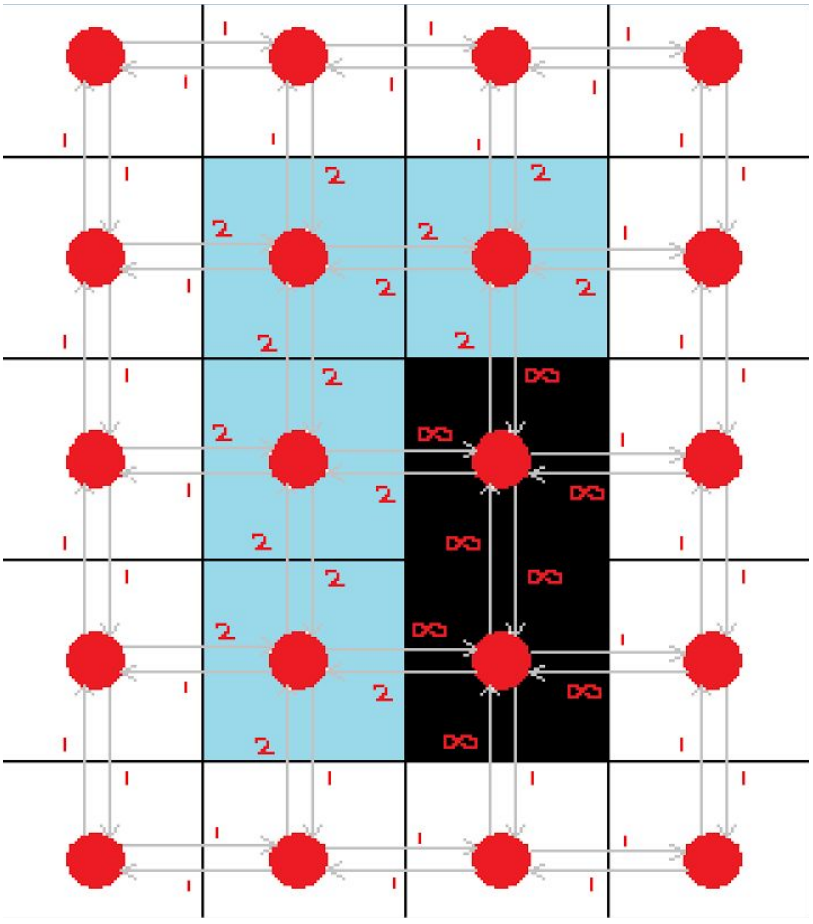
As the graph is weighted, there must be a value for each edge to indicate its weight. Take a single wall with empty tiles adjacent to it:



As a wall is defined to be impassable, all edges connected directly to the wall in the direction of the wall will have a weight of infinity. This will still be the case if the tiles adjacent to the wall are of other tile types. With a water tile, the graph will look like this:



Using this idea, we can fully construct the graph of the grid:



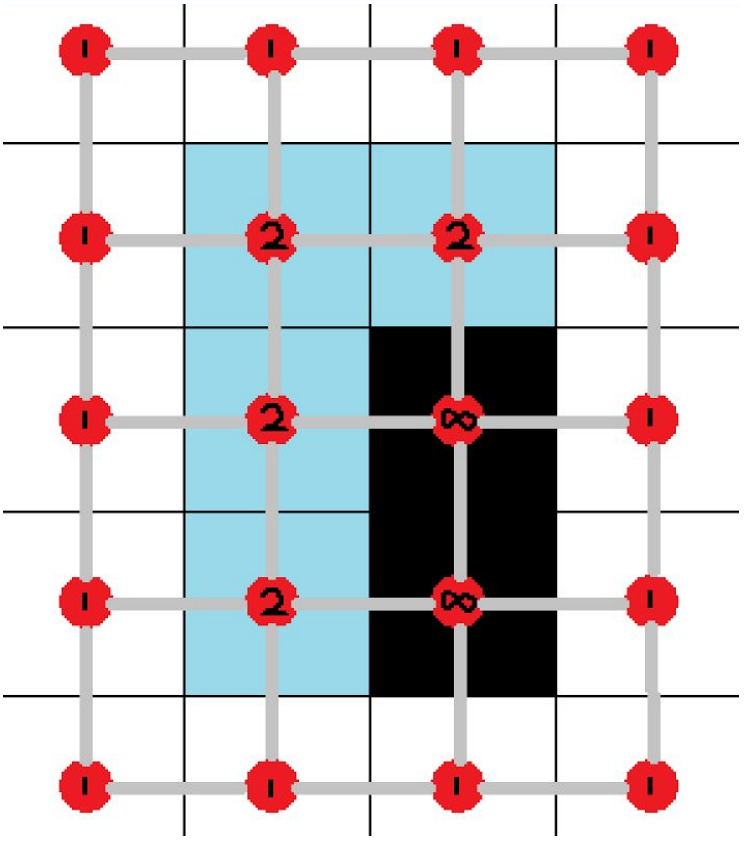
However, the diagram above is quite complicated - We'd have to store not only the weights of each edge, but also their corresponding directions.

We can store all of this information more simply by simply defining a tile type as having a weight, rather than the edges having a weight. This means when executing pathfinding algorithms, we can find the weight of the edge by simply looking at the tile's 'weight'.

Therefore,

> "*The weight of the edges connected directly to Tile A, in the direction of Tile A, is equal to the weight of Tile A.*"

We can then redraw the diagram above more compactly:



Object Oriented Planning

One possible solution to the problem involves treating each tile on the grid as an object, each of which contains a tile type and a location on the grid. The tile type could then comprise of a colour and weight, stored as a tuple.

Let the class name be ‘Tile’.

Properties of Tile:

Access Type	Variable Name	Data Type	Description
Public	TileType	String	Stores the tile type of the tile
Public	Location	Point	Stores location of tile on the grid

When executing a pathfinding algorithm, or when drawing the grid, the weight and colour needs to be extracted from the tile. We can do this by first locating the tile whose colour and or tile we need through its location. The tile’s TileType tuple can then be separated into its weight and colour by searching the TileType dictionary and then looking at the weight and colour.

Methods of Tile:

Access Type	Method Name	Parameters	Description
Public	getWeight	Location	Returns colour of tile
Public	getColour	Location	Returns weight of tile

An advantage of this implementation is that when editing tiles, we immediately know the location of the edited tile. As a result, we can quickly update the grid appropriately and we would only have to re-draw that single tile instead of the entire grid.

Additionally, since we know the tile type of the tile, we also know the weight of it which will be helpful when it comes to pathfinding - The weight of the edge connecting two adjacent tiles is equal to the weight of the next tile. Through this implementation, we can immediately find the weight of the next tile and calculate the shortest path accordingly.

Design

Terminology

Below is a list of keywords that will be used frequently in this section, as well as in further sections:

Map - The entire tile map, each tile being stored in a map list

Grid - The visible portion of the map.
(In initial designs, a ‘Grid Frame’, which is a frame that determines which part of the map to display, was considered. However, the current design does not support partial map view, so the grid and map is equivalent and used interchangeably)

Screen - An object that holds the grid

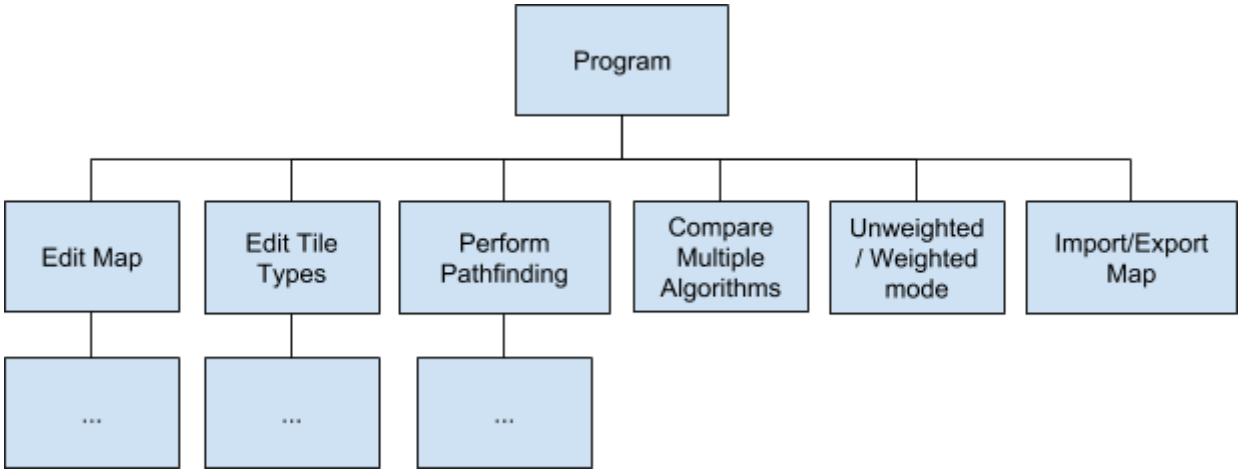
Overlay - An initially transparent screen located above the screen that holds temporary graphics such as the mouse hover highlight, path points, and tile status on pathfinding

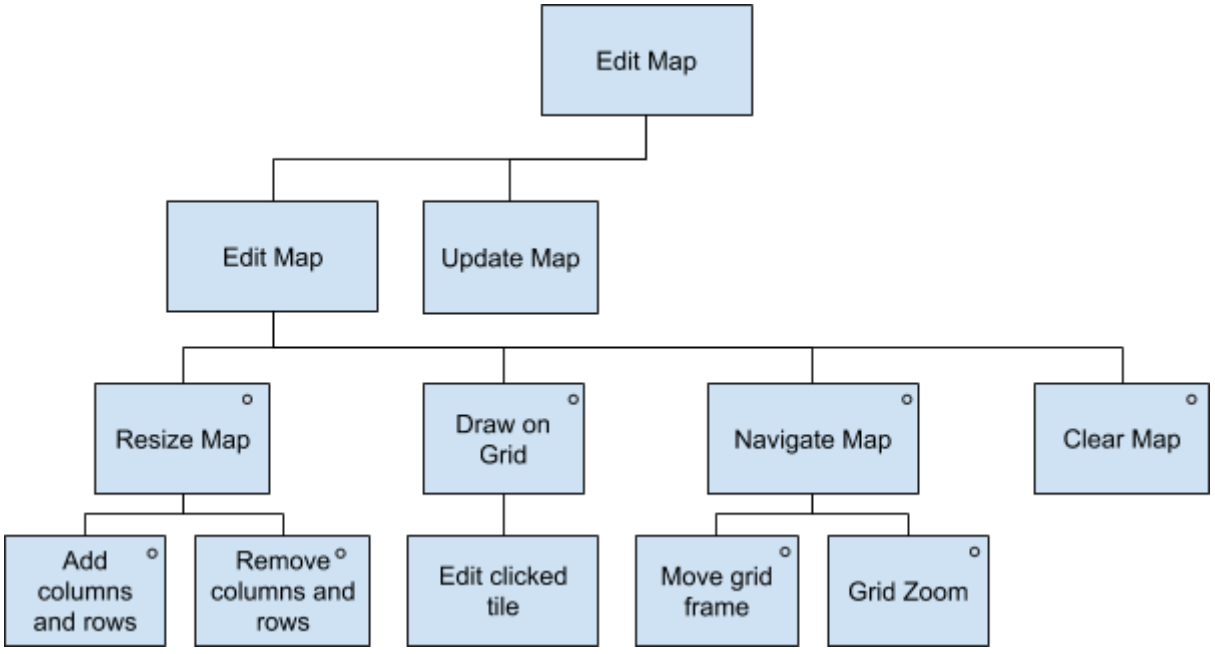
Tile Type - A particular type of tile that has a name, weight, and colour

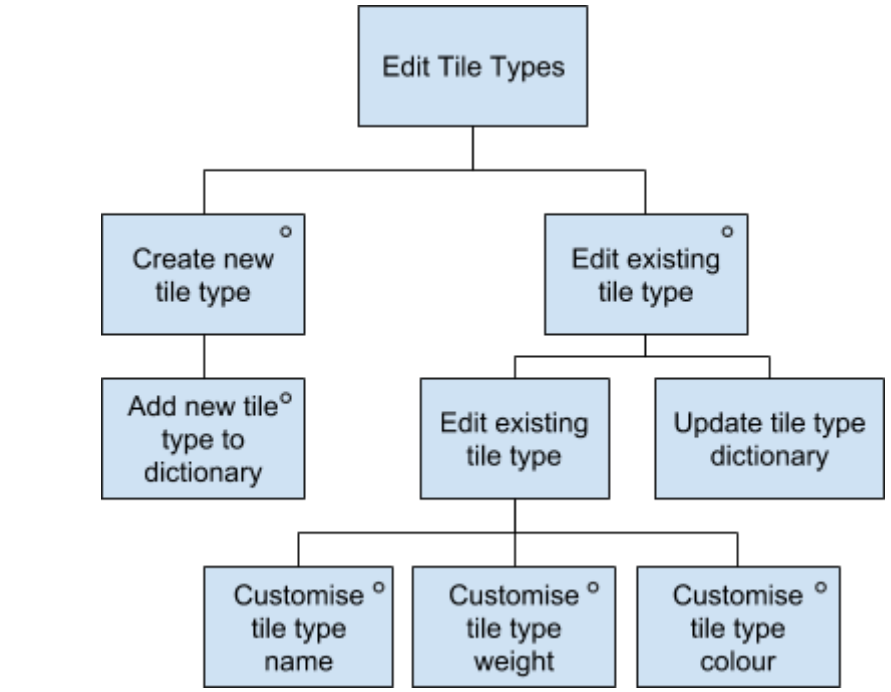
Path Points - Tiles in the map to pathfind between

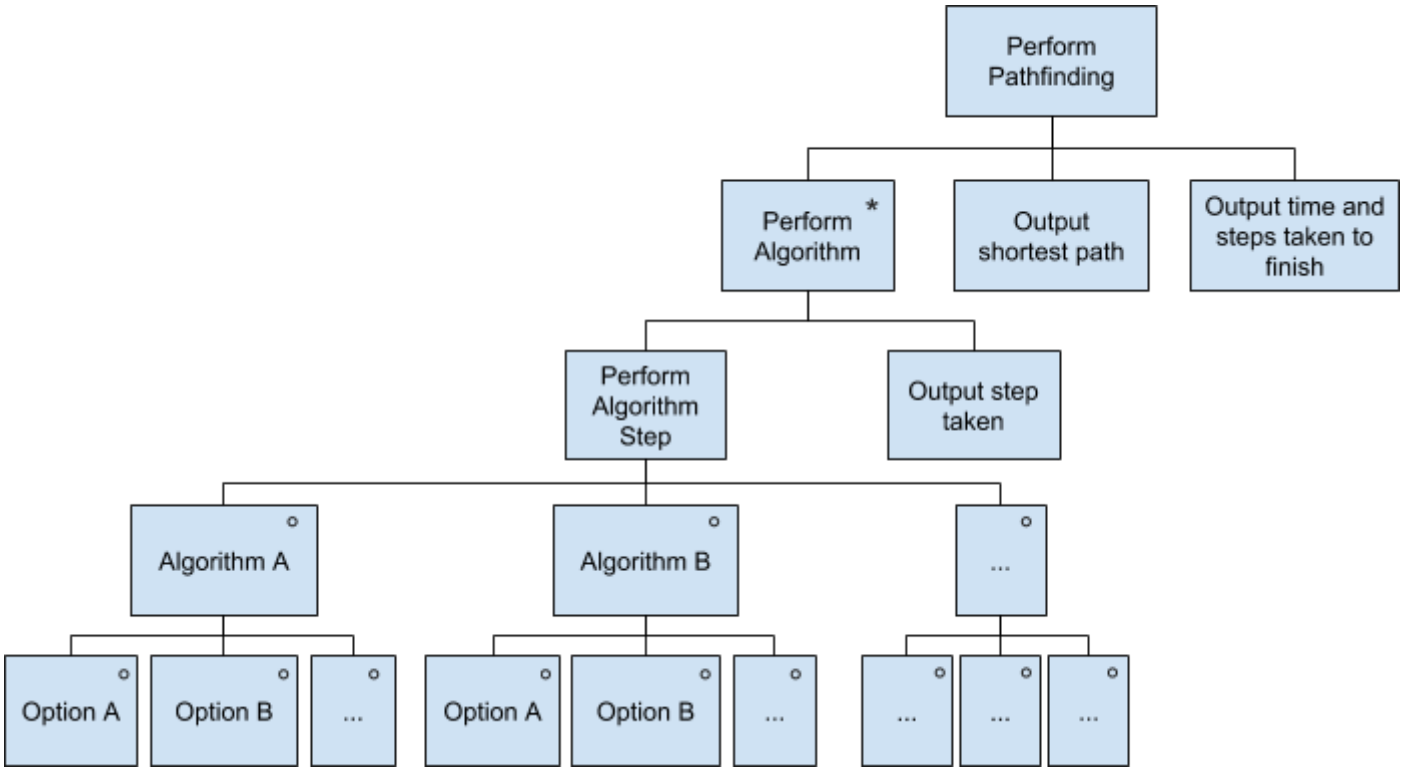
High-Level Overview

Jackson Structure Diagrams of the program have been drawn below. Due to lack of space, parts of the diagram (labelled '...' in the first diagram) have been drawn in a separate area. These can be found further below.









(Note that for the diagram above, blocks labelled '...' indicate a continuation in pattern.)

Parts of the program that have not been described by the hierarchy diagrams above are briefly described below. These parts will be explained in detail further on in the Design section:

Comparing Multiple Algorithms - Make copy of map. Number of copies depend on user input. For each map copy, have an algorithm selector and customiser. Play/pause button will be global and apply to selected algorithms in all map instances. Note that each ‘copy’ of the map really refers to the exact same map. In other words, the same map is being drawn multiple times but at different locations. Edits done to one map instance will therefore affect all other instances.

Weighted / Unweighted mode - In unweighted mode, treat weighted tiles (not empty and not wall) as walls. Note that the actual map won’t be altered. For each algorithms, adding an additional if-then-else statement when checking a tile’s type should allow for unweighted mode.

Step-by-step mode - After each iteration in the ‘Perform Algorithm’ block on the last hierarchy diagram above, the program should wait for the user to press a ‘next step’ button (or equivalent) before proceeding with the next iteration. This option can be disabled in the middle of the algorithm(s) executing. However, as the purpose of this mode is to see how an algorithm works, there will not be an ‘undo’ feature.

Import map - Create text file, write onto it the tile types and map matrix.

Export map - Reads a text file, imports tile type data and map data.

Save File

A possible save file format is as shown below:

```
2=Tall Grass=#FF00FF00,3
3=Water=#FF0000FF,4
4=Sand=#FFC2B280,2

0 0 0 0 0
0 0 1 1 0
0 0 0 2 0
0 0 2 0 0
4 0 0 3 0
```

On the top contains the name of the tile type and its colour and weight. A unique identifier is given to each tile type for brevity when referring to the tile type on the map matrix below. Each line follows the format below:

unique_id=*name*=*colour*,*weight*

Where:

Colour is in the hex format #AARRGGBB
A weight of '-1' represents an impassable tile

When reading the save file, for the first section of the file, each line is first split by the '=' character and stored in an array with its indices labelled below:

0 - *unique_id*
1 - *name*
2 - *colour*,*weight*

Tile types are created by adding the key (*name*) and its values (*colour,weight*) into the dictionary, TileType. A temporary dictionary holding *unique_id* as a key and *name* as its value is also created. This is done so each element of the map matrix can be translated into its corresponding tile type name and compared with the TileType dictionary. Note that unique_ids 0 and 1 are reserved for the default tileTypes.

This process repeats for each line until the file reader locates a blank line. Below this line contains the map matrix with each element separated by an empty space. Rows are separated by a new line. Each element is read, translated into its tile type name, and a new tile object is created with its properties assigned. There should also be a counter within the code that reads the file which keeps track of the row and column of the matrix. Each tile object is stored in an array at the index equal to its location on the map.

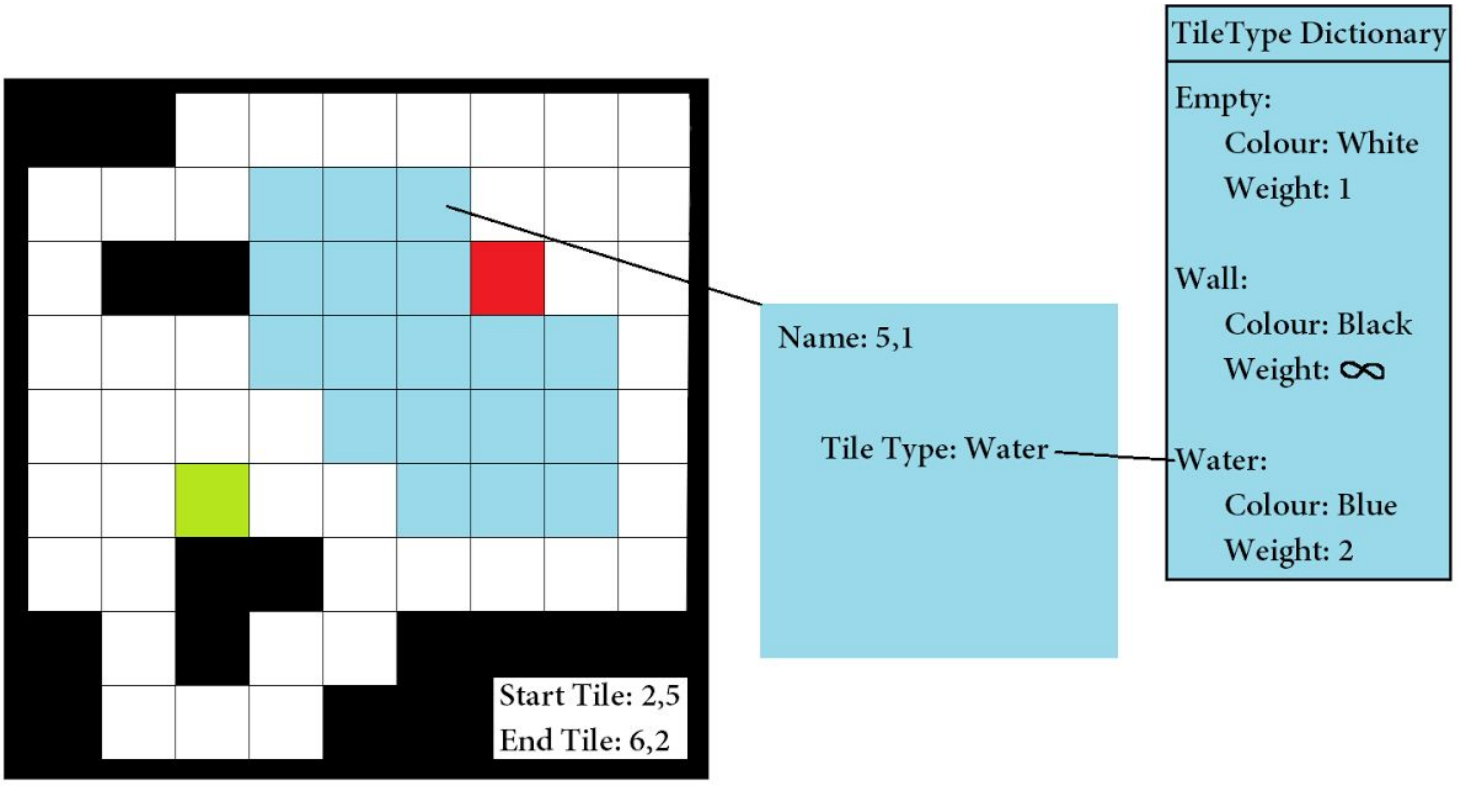
This file format has been chosen as it is easily readable by a human. This also means that users may create their own custom maps outside of the program using this save file format if they wish. They can then import the file into the program and use it for pathfinding analysis or further editing.

Description of Data Structures

Map Storage

Traditionally, to describe graphs, an adjacency matrix or adjacency list is typically used. This is not required, however, as the rule stated in the ‘Grid Design’ section of the ‘Analysis’ section allows an easier representation of the edge weights.

One possible solution is to store the map in a list of lists, each element being a tile. A ‘Tile’ class can be created to represent each of these tiles on the map. Each tile will have a name that corresponds to its location on the map and also a property, ‘TileType’, which will indicate what type of tile it is. The value for TileType can be referred to a dictionary that stores the name of the tile type, as the key, and its colour and weight, as the key’s values. The values of the key can be stored as a tuple:



The reasoning for the choice of the above data structures is stated below:

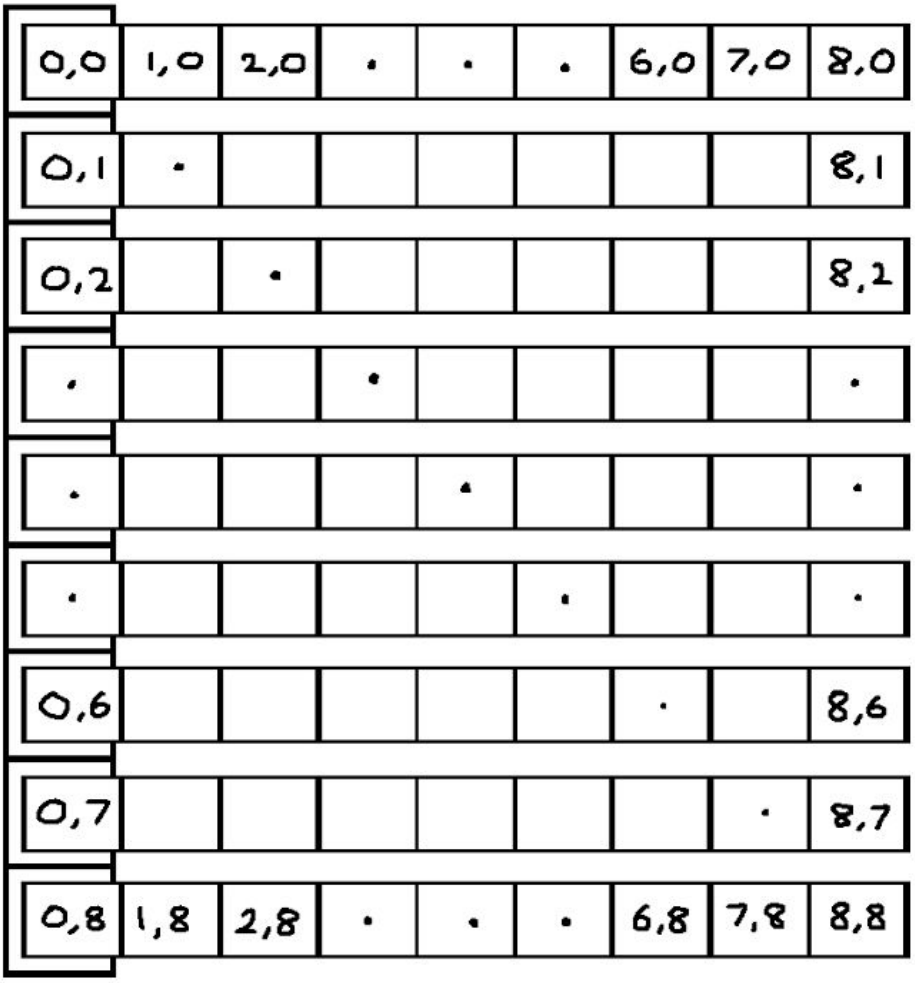
> As the map will have to be resized, a dynamic data structure must contain it. Thus, a list of lists is used rather than an array. Having a list of lists rather than just a single list will allow us to treat the structure like a 2D array. It also reduces the complications involved when resizing the map and having to change the column and row data within the list.

A TileType dictionary is created so that when editing the weights of tiles, only one part of the program needs to be edited and not every single tile on the map.

A Tile class is created to link the map with the TileType dictionary. The name of each tile represents its coordinates on the map and its Tile Type holds a string value to be referred to the TileType dictionary. The “getWeight” method can be called to return a specific tile’s weight, which will be useful during pathfinding.

A tile type must have a colour and weight. The name of the tile type is the dictionary’s key but the dictionary’s value can only hold one value. A tuple can therefore be used to combine the colour and weight together. It can also be separated later on when needed.

A list of lists can be used to represent the map above. A diagram of this data structure is shown below:



TileType Storage

In addition to storing the map, we need to also be able to store information on tileTypes. Two dictionaries will be used for this - dicTileTypeInfo and dicTileTypeTiles:

dicTileTypeInfo will store the colour and weight of a tileType. These two values will be stored in a tuple and this tuple will be stored as the dictionary's value. This is needed as dictionaries can only store one value.

dicTileTypeTiles stores a list of tiles on the map that is of a certain tileType. These tiles will be stored in a HashSet, which is then stored as the dictionary's value. The reason for using a HashSet is due to its quick lookup time of O(1) and also due to HashSets only storing one value per element. This which will be useful when quickly editing a large amount of tiles at once.

Dictionaries have been used mainly due to their quick lookup times of O(1). Each tileType has a unique name which is then be mapped onto a list of unique dictionary keys. By having both dictionaries share the same key, I will be able to access all the information about a tileType that I need just from the name of that tileType. For both dictionaries, the name of the tileType will be the key of each dictionary entry. A list, lstDicTileTypeKeys, will be used to store these names.

Pathfinding

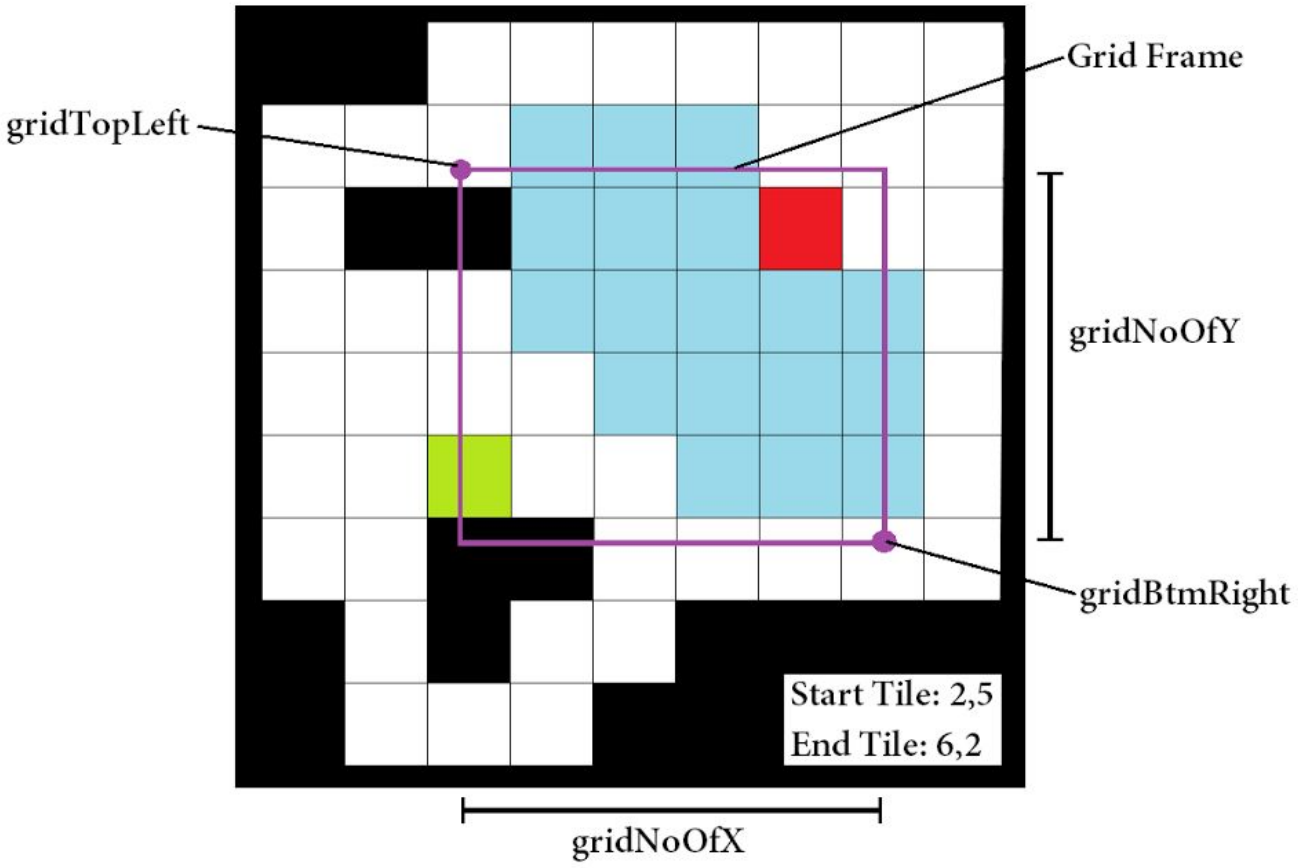
A custom class, 'PriorityList', will be used to hold a list of tiles to be checked when pathfinding with Dijkstra's algorithm and all variants of the A* algorithm. A class diagram of this can be found in the 'Classes' section further below the design section.

This class is an extension of a the abstract data type, List, where the elements of is list is sorted based on some value - The PriorityList for Dijkstra's algorithm has its elements sorted based on their tentative distance whereas the A* algorithm's PriorityList is sorted based on the element's F-Score.

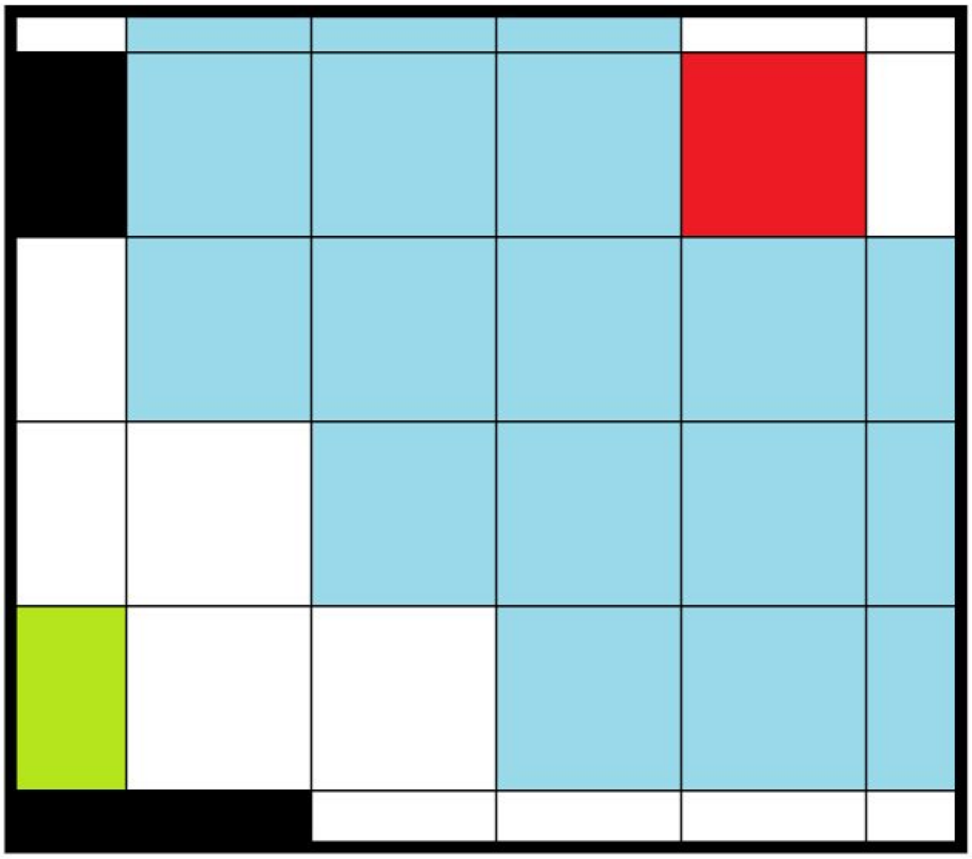
When a shortest path is found and the path is to be output to the screen, the path is backtracked from the destination tile to the source tile and each tile that is met during this process is pushed into a stack. Once the source tile is reached, the stack continuously pops its tiles and marks the tile on the screen as a path until the stack is empty. This reverses the order that tiles are backtracked and makes it seem as if the path is being traced from start to end. This reversal is the reason why a stack is used.

Grid Frame (Obsolete)

A grid frame encases the image that will be displayed in the picturebox to the user. It is used so we can display a subset of the map's elements. It comprises of a top-left and bottom-right tile's coordinate, and the number of tiles to display in a column and a row. These variables will determine which part of the map to display on the grid:



Tiles within the Grid Frame will be displayed on-screen (grid). Below is an image of this:



Note that the grid frame does not necessarily sit on a tile's border. It can move around freely in between tiles.

The top-left and bottom-right corners of the grid frame are need for movement and zooming. The variables that store this information are 'gridTopLeft' and 'gridBtmRight' respectively.

Pseudocode

Best-Fit Picturebox Algorithm (Obsolete)

Drawing picturebox of best fit:

```
testLengthX = Floor((pnlGridWidth - 1) / noOfTilesX)
testLengthY = Floor((pnlGridHeight - 1) / noOfTilesY)

If testLengthX >= testLengthY
        tileLength = testLengthY
If testLengthY > testLengthX
        tileLength = testLengthX

pbxGridWidth = tileLength * noOfTilesX + 1
pbxGridHeight = tileLength * noOfTilesY + 1
```

X (Cross) limit:

```
noOfTilesXY = Floor((pnlGridXY - 1) / 4)
```

Grid line limit:

```
noOfTilesXY = Floor((pnlGridXY - 1) / 2)
```

Tile limit:

```
noOfTilesXY = pnlGridXY - 1
```

Out of bounds:

```
noOfTilesXY > pnlGridXY - 1
```

Minimum:

```
noOfTilesXY = 1
```

(Note: ‘Grid line limit’ is the minimum size at which both the tile and its borders can be displayed. ‘Tile limit’ is the size at which one tile occupies one pixel, which is too small for tile borders to be displayed properly.)

Add New TileType

Algorithm for when the add button, '+', in tabTiles is clicked

```
/* ALGORITHM:
    * Create new panel, put in correct location, and add required controls into it
    * Create new tileType
    * Increment panel counter by 1 (and also for unset panels)
    * Move pnlAddSet down to prepare for next new panel
    */
   //Note: Click on panel (not on its controls) to select a tileType to place
   //      Click on pbx in panel to change tile colour
```

Set TileTypes

Algorithm for when the 'Set' button in tabTiles is clicked

```
/* ALGORITHM:
 * For each custom panel in list
 *     If pass (Validation Check)
 *         Recreate dictionaries
 *         Update list key name
 *         Update panel name
 *         Change panel backcolour to green
 *         Recolour tiles of updated tileType
 *     Else
 *         Change panel backcolour to red
 *
 * If all panels green
 *      Map ready to edit
 */
```

(**Note:** For panel background colours,
Red = Invalid, Yellow = Pending, Green = Valid)

Delete TileTypes

Algorithm for when the delete button, ‘X’ on a tileType panel is clicked

```
 /* ALGORITHM:
        *  Notify user that tiles of this type on grid will be deleted
        *If 'OK',
        *   Delete tiles from map (if any) - Set to default
        *   Obtain indexToDelete by [(btnDelete.Y - 1stpnl.TopLeft.Y) / pnlHeight]      {1st
pnlTopLeft = 190}
        *   Move pnlAddSet up
        *   Delete panel (parent of btnDelete) from tabTiles
        *   Decrement panel counter by 1
        *   Use indexToDelete to delete key from dictionary
        *   For all index higher than (index to delete) in list
        *       Move panel upwards in tabTiles
        *       Rename unset panels that have moved and recreate in dictionary
        *       Move panel index in list by 1 towards 0
        *   Delete last element of list of keys (lst.Count - 1)
        *   Determine if btnSet will be visible or not
        */
```

dicTileTypeTiles

Governs how the dictionary of tiles of certain tileTypes will operate

```
/* FOR LIST OF TILES IN MAP OF CERTAIN TILETYPES:
 * Changing noOfTilesXY:
 *      For deleted tiles - Get Tile coords, delete from HashSet
 *      For added tiles - Add Tile coords to HashSet
 *
 * (Note: Don't create dictTileTypeTiles entry until tileType is confirmed set)
 *
 * Updating tileTypes:
 *   If btnSet clicked
 *      If userinput valid
 *          Create new empty HashSet
 *          If there is previous entry in dicTileTypeTiles
 *              Move HashSet into new empty HashSet
 *              Delete dictionary entry
 *              Update existing tiles on map - Change tileType value to new value
 *          Create dictTileTypeTiles entry and move in HashSet
 *      Redraw updated tiles on grid
 *
 * Deleting tileType:
 *      If entry exists for tileType to delete in dicTileTypeTiles (tileType previously set)
 *          Delete tiles on map - Set tileType to "Empty"
 *          Redraw updated tiles on grid
 *          Remove dicTileTypeTiles entry completely
 *
 * Edit Tile on map:
 *      Get tileCoords and old tileType
 *      Remove current tile from old tileType HashSet in dicTileTypeTiles
 *      Update tileType
 *      Add current tile to updated tileType HashSet in dicTileTypeTiles
 */
```

Map Rebuild

Algorithm for resizing map

```
//MAP REBUILD PSEUDOCODE - Row = Y, Column = X
        /* If more rows than original
         *     For each new row
         *         Add new no. of columns (tiles)
         *         Add each new tile to set of tiles of some tileType
         * Else if less rows than original
         *     For each extra row
         *         Delete each tile of this row from dicTileTypeTiles
         *         Delete extra row
         * End If
         *
         * If more columns than original
         *     Add new columns (tiles) to original rows that are still within map
         *     Add each new tile to set of tiles of some tileType
         * Else if less columns than original
         *     For each original row that are still within map
         *         Delete extra columns (tiles)
         *         Delete each of these tiles from dicTileTypeTiles
         */
```

Breadth-First Search Algorithm

```
/* BFS Algorithm:
 *
 * Add start point to queue tilesToCheck
 * Add start point with layerNo = 0 to dictionary tilesSeen
 *
 * While tilesToCheck NOT empty
 *          currTile = tilesToCheck.Dequeue
 *          Mark currTile as 'Working' and display on overlay
 *
 *          For each nextTile adjacent to currTile
 *          If nextTile within bounds, not marked as seen, and has 'Empty' tileType
 *          If nextTile = endPoint
 *                    Clear tilesToCheck
 *                    pathFound = True
 *          Else
 *                    tilesToCheck.Enqueue(nextTile)
 *                    Mark nextTile as 'Seen' and display on overlay
 *
 *          Add nextTile with its layerNo into tilesSeen
 *
 *          Mark currTile as 'Seen' and display on overlay
 * If pathFound
 *          Trace Shortest Path
 * Else
 *          Output "Could not find path to destination"
 */

 /* Tracing shortest path [BFS]:
 *
 * layerNo = endPoint's layerNo
 * Push endPoint into shortestPath Stack
 * currTile = endPoint
 *
 * While layerNo > 0
 *          For each nextTile adjacent to currTile
 *          If dictionary tilesSeen contains currTile as key
 *          If layerNo of currTile = layerNo - 1
 *                    Push currTile into shortestPath
 *
 * While shortestPath NOT empty
 *          pathTile = shortestPath.Pop
 *          Mark pathTile as 'Path' and display on overlay
 */
```

Improved Best-Fit Picturebox (Best-Fit Screen) Algorithm

Updated initial best-fit picturebox algorithm for multiple screens:

```
testLengthX = Floor((pnlGridWidth - 1) / noOfTilesX)
testLengthY = Floor((pnlGridHeight - 1) / noOfTilesY)

If testLengthX >= testLengthY
        initTileLength = testLengthY
If testLengthY > testLengthX
        initTileLength = testLengthX

If initTileLength < minTileLength
        belowMinTilelength = true
Else
        slotLength = (noOfTilesX * testTileLength) + borderLength + 1
        layerLength = (noOfTilesY * testTileLength) + borderLength + 1
        slotsPerLayer = Floor(pnlGridWidth / slotLength)
        noOfLayers = Floor(pnlGridHeight / layerLength)
        totNoOfSlots = slotsPerLayer * noOfLayers

        While totNoOfSlots < noOfScreens
                slotTileLength = Floor((1 / noOfTilesX) * ((pnlGridWidth /
(slotsPerLayer + 1)) - borderLength - 1))
                layerTileLength = Floor((1 / noOfTilesY) * ((pnlGridHeight /
(noOfLayers + 1)) - borderLength - 1))

                If slotTileLength > layerTileLength
                        tileLength = slotTileLength
                        totNoOfSlots += noOfLayers
                        slotsPerLayer++

                ElseIf layerTileLength > slotTileLength
                        tileLength = layerTileLength
                        totNoOfSlots += slotsPerLayer
                        noOfLayers++

                ElseIf slotTileLength = layerTileLength
                        tileLength = slotTileLength
                        totNoOfSlots += noOfLayers
                        slotsPerLayer++
                        totNoOfSlots += slotsPerLayer
                        noOfLayers++

                If tileLength < minTileLength
```

```
                belowMinTilelength = true

        If belowMinTilelength
            Reject tileLength
        Else
            slotLength = (noOfTilesX * tileLength) + borderLength + 1
            layerLength = (noOfTilesY * tileLength) + borderLength + 1
            slotX = 0
            slotY = 0

            For each screen in listOfScreens
                screen.Location.X = slotX * slotLength
                screen.Location.Y = slotY * layerLength
                screen.Width = slotLength
                screen.Height = layerLength
                Add screen to pnlGrid

                slotX++
                If slotX = slotsPerLayer
                    slotY++
                    slotX = 0
```

Also, there is no need to keep track of cross limit, grid line limit, tile limit. This is replaced by simply checking if the resulting tileLength is less than a given minimum value:

```
minTileLength = 4
```

The above value includes one row and column for each tile being used up due to the grid lines

How the algorithm works

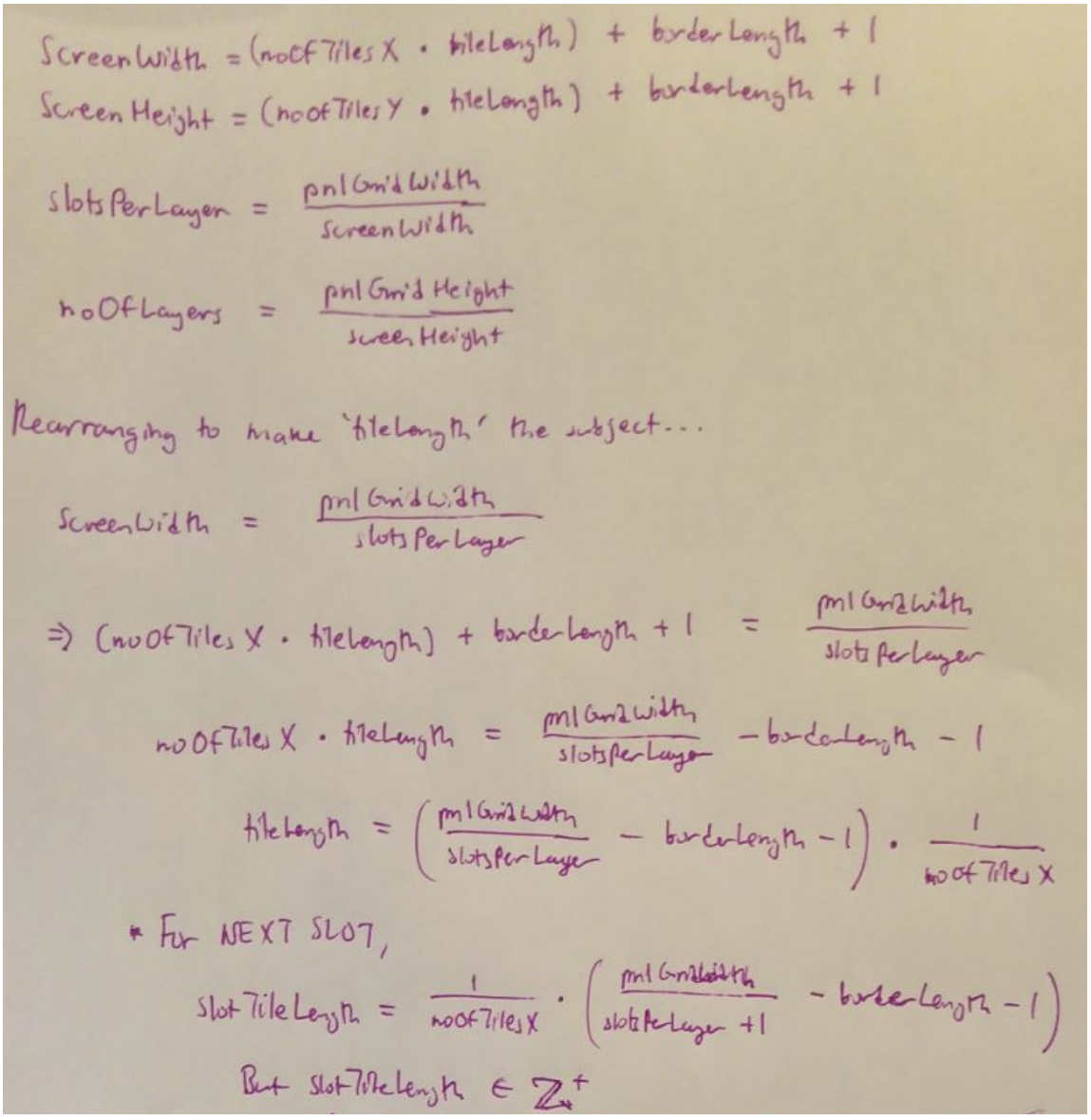
Conceptually, the algorithm works by gradually decreasing the tileLength from some initial value (where only one screen exists). If there is initially no space for another screen, then eventually, one of three things will happen:

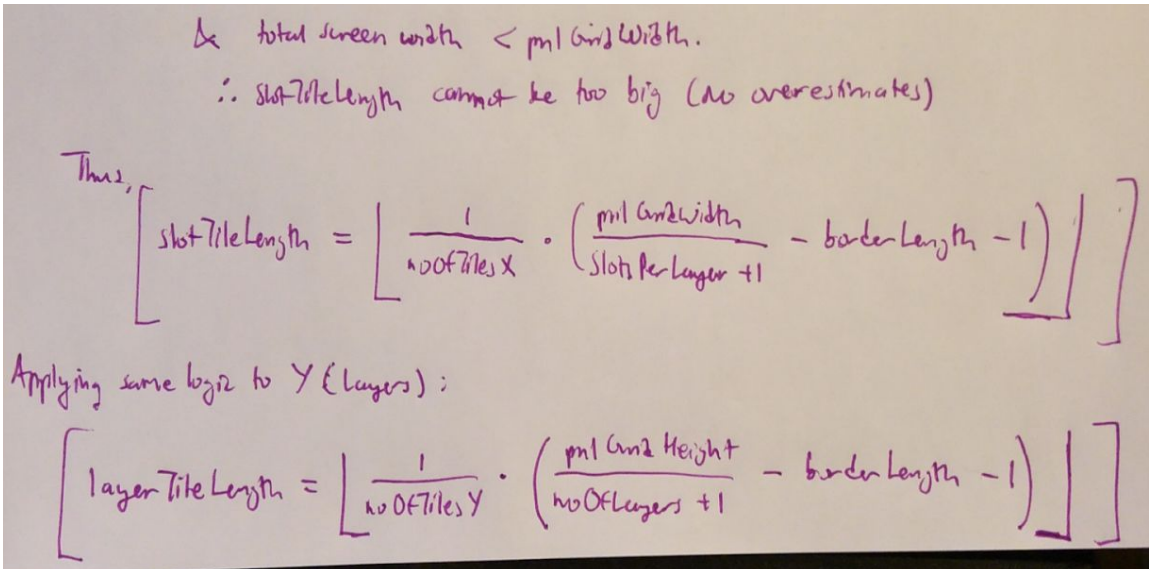
- There will be enough space for a new screen to the left of the existing screens (X).
- There will be enough space for a new screen on the bottom of the existing screens (Y).
- There will be enough space for a new screen in both X and Y (XY).

If state XY does not happen, then state X and state Y will happen at different tileLength values. The algorithm determines which state will occur first and then decides if there is enough space for all screens to fit. If there isn't, then this process repeats until there is enough space.

This ‘space’ for a screen is called a ‘slot’. A row of slots is called a ‘layer’. The algorihtm stops creating slots when the total number of slots sufficient to fit all screens into pnlGrid. The algorithm will place the screens into these slots in the order left to right, top to bottom. If at any stage of the process the minimum tileLength value is reached, the algorithm will revert the state change that caused algorithm to run in the first place.

Derivation of formulae




∴ total screen width < pnl Grid Width.
∴ slotTileLength cannot be too big (∴ overestimates)
Thus,
slotTileLength = ⌊ 1 / noOfTilesX · (pnl GridWidth / Slots Per Layer + 1 − borderLength − 1) ⌋
Applying same logic to Y (Layers):
layerTileLength = ⌊ 1 / noOfTilesY · (pnl Grid Height / noOfLayers + 1 − borderLength − 1) ⌋

Dijkstra’s Shortest Path Algorithm

(Note: The pseudocode below is built off the BFS algorithm pseudocode)

```
        /* Dijkstra's Algorithm:
        *
        * Add start point (key) with tentative distance of 0 (val) to dictionary tilesToCheck
        * Add start point (key) with tentative distance of 0 and previous connected tile of ""
(val) dictionary tilesSeen
        *
        * While tilesToCheck NOT empty
        *       currTile = Node with least tentative distance
        *       Remove this node from dictionary tilesToCheck
        *       Mark currTile as 'Working' and display on overlay
        *
        *       For each nextTile adjacent to currTile
        *       If nextTile within bounds, not marked as seen
        *       If nextTile = endPoint
        *               Clear tilesToCheck
        *               pathFound = True
        *       Else
        *               nextTileDist = currTile's tentDist + nextTile's weight
        *               tilesToCheck.Add(nextTileCoords, nextTileDist);
        *               Mark nextTile as 'Seen' and display on overlay
        *
        *               Add nextTile with nextTileDist and currTile (as previous connected tile)
into tilesSeen
        *
        *       Else If tile previously seen
        *       Mark nextTile as 'Updating' and display on overlay
        *       Compare nextTile's current tentative distance with nextTileDist
        *       Update nextTile's tentative distance if nextTileDist is less than the current
value
        *       Mark nextTile as 'Seen' and display on overlay
        *
        *       Mark currTile as 'Seen' and display on overlay
        *
        * If pathFound
        *       Trace Shortest Path
        * Else
        *       Output "Could not find path to destination"
        */
```

```
/* Tracing shortest path [Dijkstra]:
*
* Push endPoint into shortestPath Stack
* currCoords = endPoint
*
* While currCoords != startPoint
*         prevCoords = currCoords' previous coordinates from tilesSeen
*         Push prevCoords into shortestPath
*         currCoords = prevCoords
*
* While shortestPath NOT empty
*         pathTile = shortestPath.Pop
*         Mark pathTile as 'Path' and display on overlay
*/
```

A* Algorithm

(Note: The pseudocode below is built off the Dijkstra's algorithm pseudocode)

```
/* A* Algorithm:
*
* Add start point (key) with G-score of 0, and F-score of 0 (val) to dictionary tilesToCheck
* Add start point (key) with G-score of 0, F-score of 0 and previous connected tile of "" (val) dictionary tilesSeen
*
* While tilesToCheck NOT empty
*       currTile = Node with least F-score
*       Remove this node from dictionary tilesToCheck
*       Mark currTile as 'Working' and display on overlay
*
*       For each nextTile adjacent to currTile
*       If nextTile within bounds, not marked as seen
*       If nextTile = endPoint
*               Clear tilesToCheck
*               pathFound = True
*       Else
*               newGscore = currTile's tentDist + nextTile's weight
*
*               x = Magnitude of x-dist of nextTile from endPoint
*               y = Magnitude of y-dist of nextTile from endPoint
*               If heuristic = Manhattan
*               newHscore = x + y
*               Else If heuristic = Euclidean
*               newHscore = sqrt(x^2 + y^2)
*               Else if heuristic = Chebyshev
*               If x >= y
*               newHscore = x
*               Else
*               newHscore = y
*               newFscore = newGscore + newHscore
*
*               tilesToCheck.Add(nextTileCoords, (newGscore, newFscore));
*               Mark nextTile as 'Seen' and display on overlay
*
*               Add nextTile with newGscore, newFscore, and currTile (as previous connected tile) into tilesSeen
*
*       Else If tile previously seen
*       Mark nextTile as 'Updating' and display on overlay
```

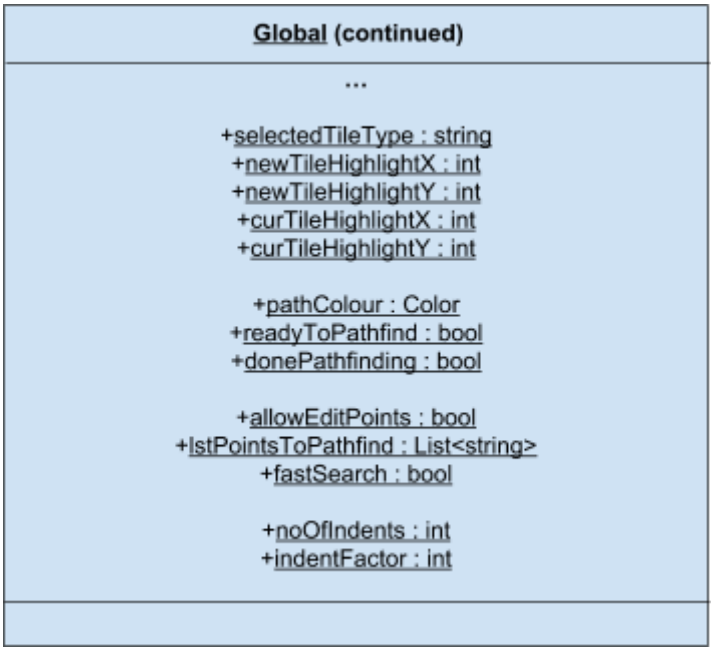
```
        *       Compare nextTile's current F-Score with newFscore
        *       Update nextTile's G-score and F-score if newFscore is less than the current
value
        *       Mark nextTile as 'Seen' and display on overlay
        *
        *       Mark currTile as 'Seen' and display on overlay
        *
        * If pathFound
        *       Trace Shortest Path
        * Else
        *       Output "Could not find path to destination"
        */

        /* Tracing shortest path [A*]:
        *
        * Push endPoint into shortestPath Stack
        * currCoords = endPoint
        *
        * While currCoords != startPoint
        *       prevCoords = currCoords' previous coordinates from tilesSeen
        *       Push prevCoords into shortestPath
        *       currCoords = prevCoords
        *
        * While shortestPath NOT empty
        *       pathTile = shortestPath.Pop
        *       Mark pathTile as 'Path' and display on overlay
        */
```

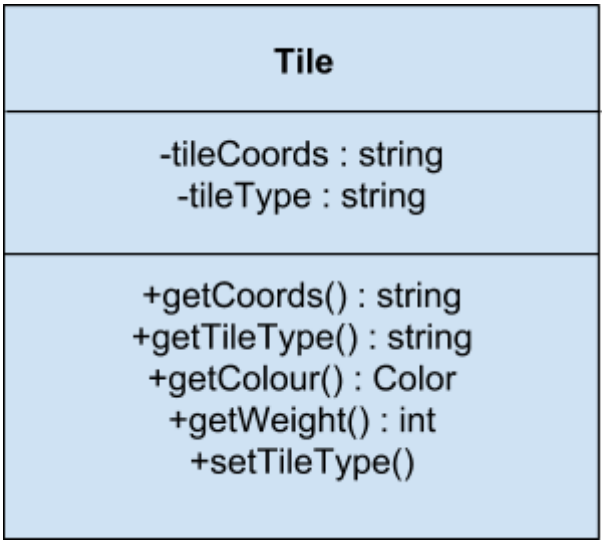
Classes

There are four custom classes in total: Global, Tile, PriorityListDijkstra, PriorityListAStar

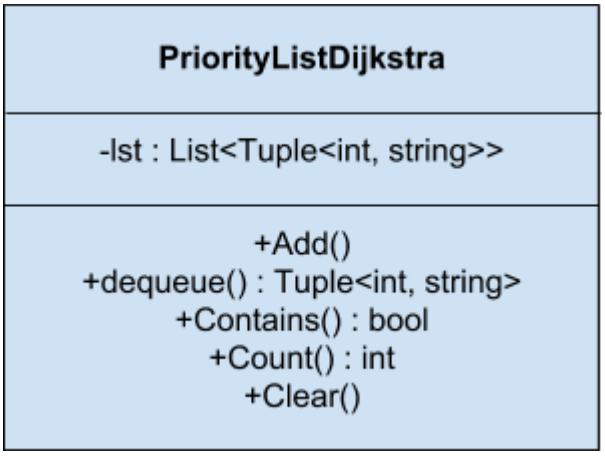
The ‘Global’ class is a static class that holds the variables that will be used throughout multiple modules of the code:



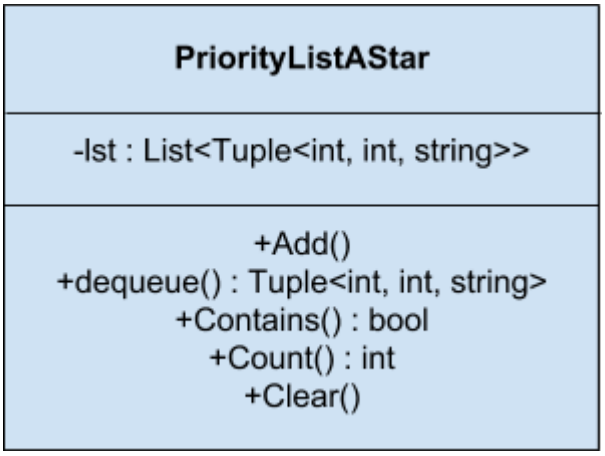
The ‘Tile’ class, as the name suggests, describes the individual tiles on the map. Each tile has a coordinate, tileCoords, and a tileType:



The ‘PriorityListDijkstra’ class is a custom list to be used in Dijkstra’s Algorithm that holds a set of tiles to be checked and sorts them based on their tentative distances:



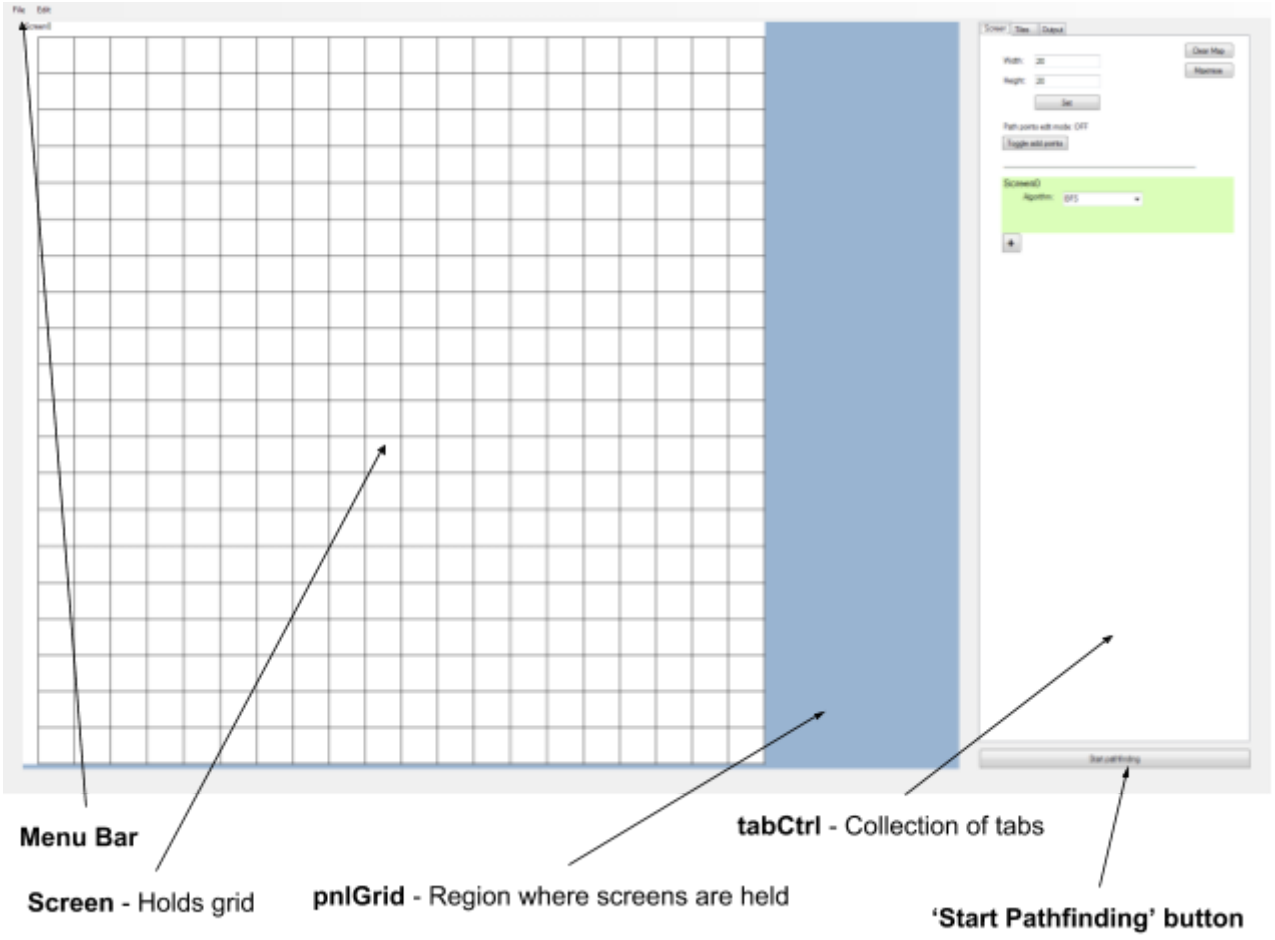
The ‘PriorityListAStar’ class is a custom list to be used in all variants of the A* Algorithm. This class holds a set of tiles to be checked and sorts them based on their F-Scores:



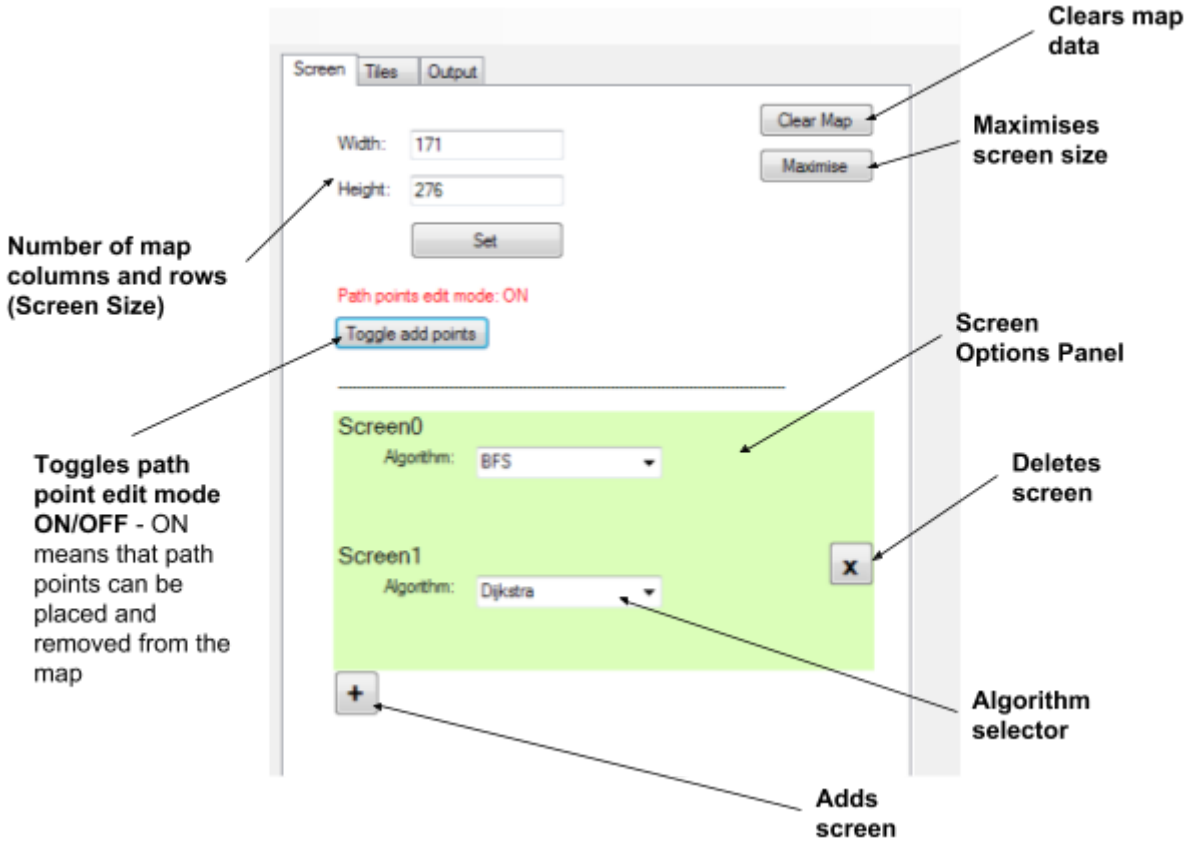
(Note: A superclass, PriorityList, has been considered for PriorityListDijkstra and PriorityListAStar. However, this idea was abandoned due to complications involving the differences in the list used in both PriorityLists and how the methods, Add, dequeue, and Contains, depends on the definition of this list.)

User Interface Design

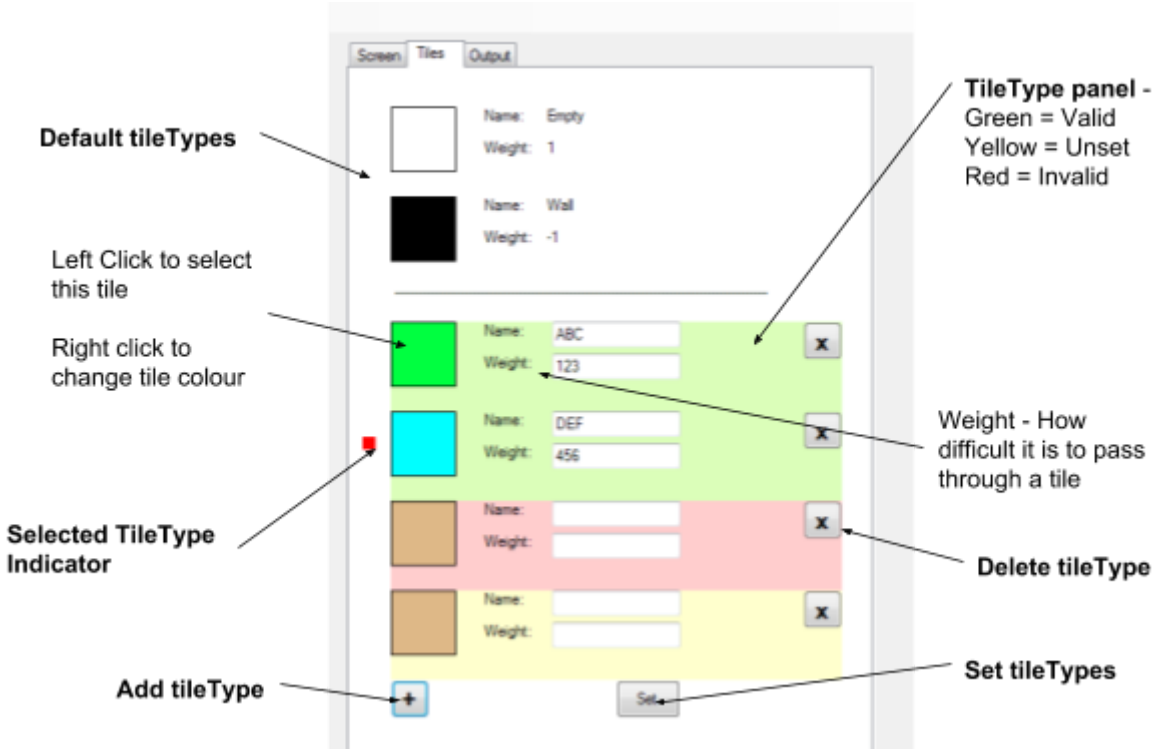
Initial screen:



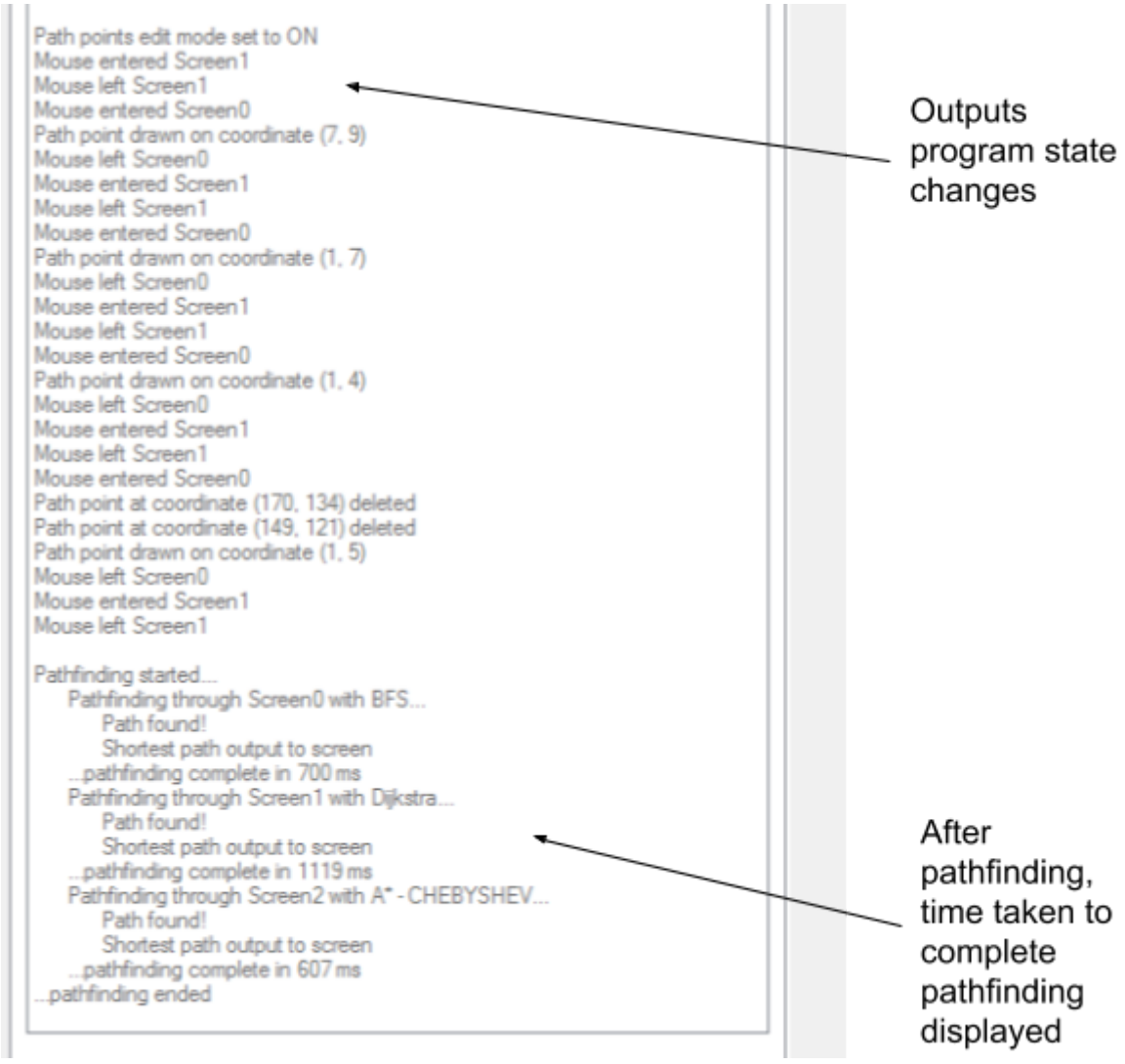
Screen tab:



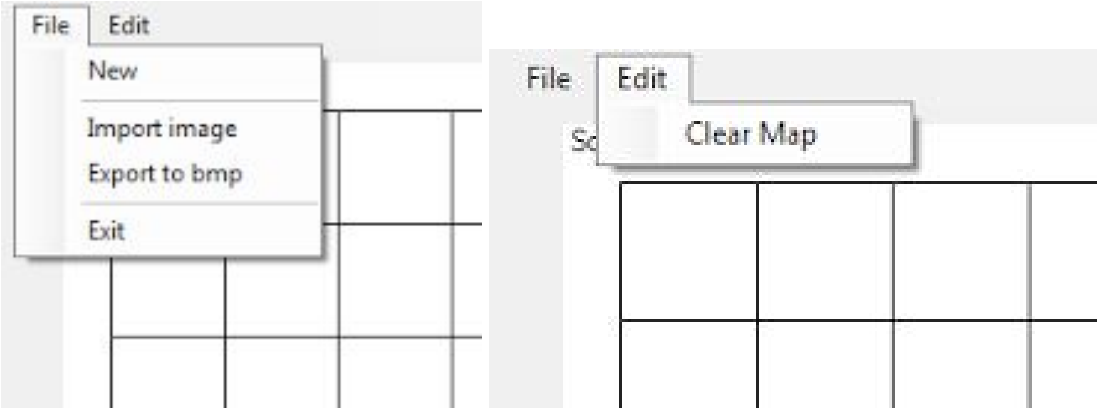
TileType tab:



Output tab:



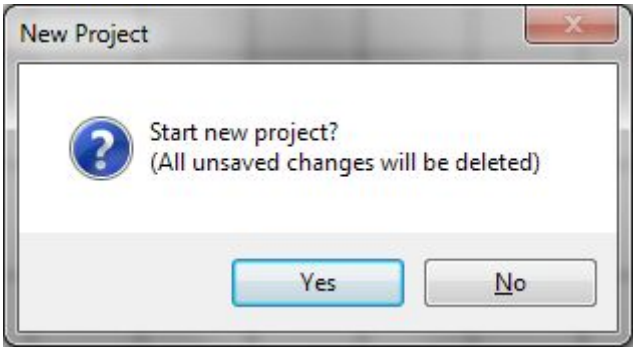
Menu Bar Options:



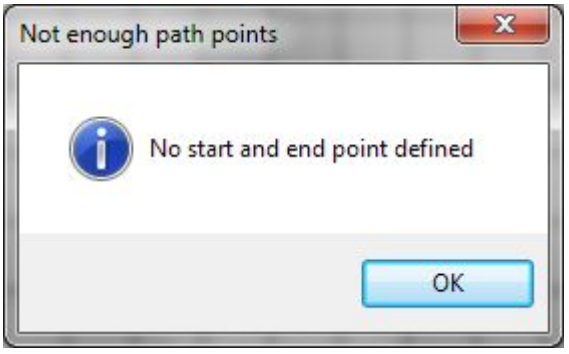
Human-Computer Interactions

There are various ways in which the user can interact with the program. Below is a list of some of the user prompts the program has to offer:

After selecting File - New Project:

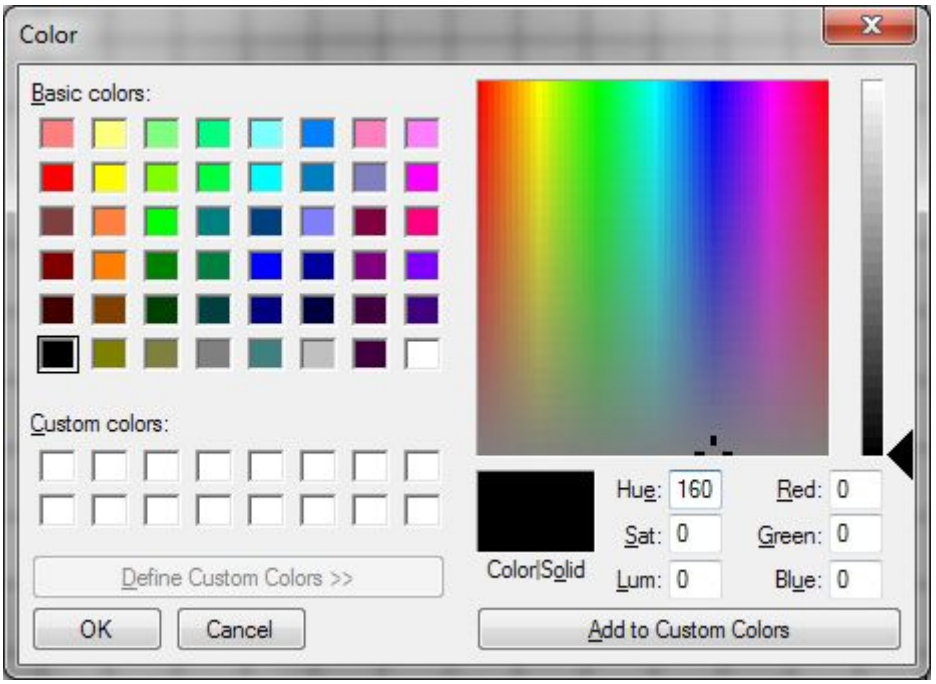


'Start Pathfinding' clicked with less than 2 path points on the screen overlay:



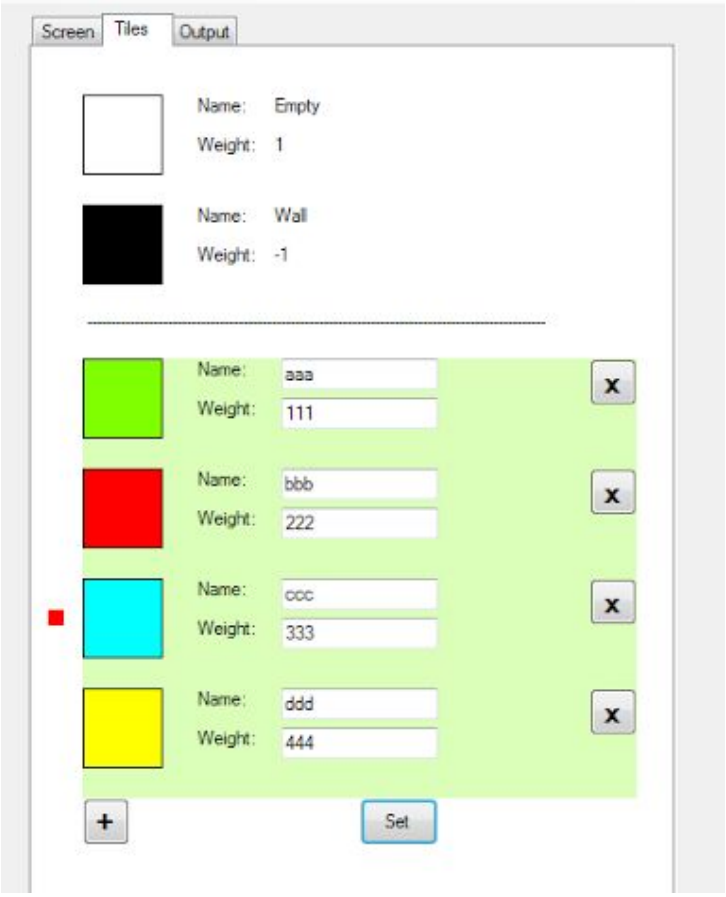
The above message boxes prevent all other processes in the program from occurring whilst it is displayed. This is so that the user can easily see the alert message and cannot do anything unexpected whilst the message is still displayed.

Right clicking on the tileType colour:



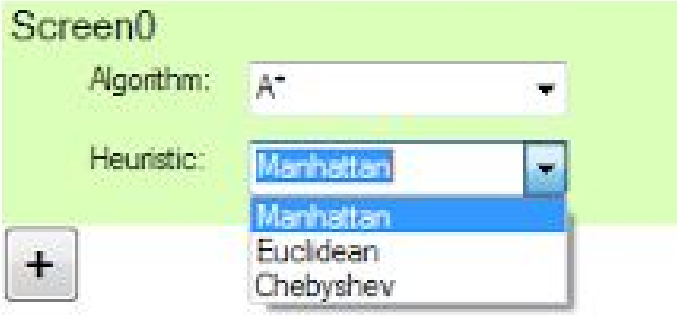
This colour dialog provides an easy way for the user to choose a colour for a tileType.

Red Box = Selected tileType indicator:



A red box has been used to show which tileType has been selected as it is clearly visible and therefore easy to tell which tileType has been selected.

ComboBox for algorithm selection for each screen:

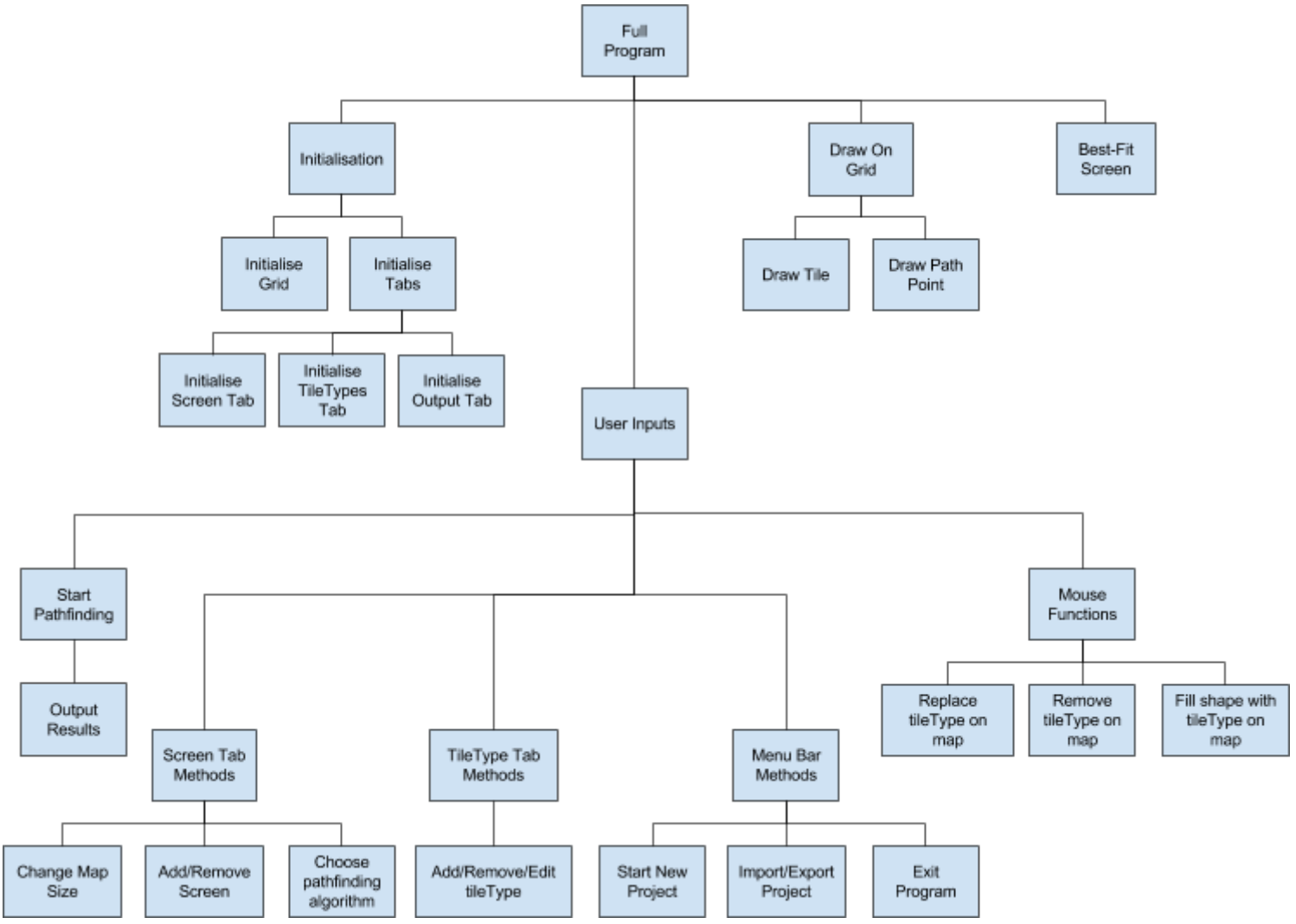


ComboBoxes make it easy for the user to choose from a selection of algorithms and heuristics (for the A* algorithm).

Technical Solution

Overview Guide

The hierarchy diagram below shows the general structure of the program code:



(The form design is the same as the User Interface Design in the design section.)

The full code has been divided into the sections:

-Classes
-Initialisation
-tabCtrlMethods
-BestFitScreen
-DrawGrid
-MouseFunctions
-UsefulFunctions
-menuBarMethods
-Pathfinding

For each of these sections, there is a brief summary of the code included. In addition to this, interesting features that the code contains, for example implementations of the algorithms mentioned in the design section or the use of more sophisticated programming techniques, will be mentioned.

Comments have been written alongside the code to help make understanding it easier. However, note that in many of the subroutines below, there is the unexplained code:

```
string tbxOutput = "";
tbxOutput = string.Format("Resetting project...");
outputToTbxOutput(tbxOutput);
```

This code is for outputting messages to the project log, located in tabOutput. Since it is used so frequently, I did not comment on its purpose in the code.

Classes

The code below is an implementation of the classes described in the design section:

```
#region Classes
       public static class Global        //GLOBAL VARIABLES
       {

//CONSTS-------------------------------------------------------------------------------
------------------
              public const int maxTileTypes = 30;      //Excludes default tileTypes

              public const int maxScreens = 20;
              public const int borderLength = 23;
              public const int screenOptionsHeight = 85;
              public const int minTileLength = 4;             //Includes +1 from
gridline

              public const int maxPathPoints = 5;


              //Indicator colours? - valid, working, invalid
              //Default colours? - burlywood


//MAP----------------------------------------------------------------------------------
------------------
              public static List<List<Tile>> lstMapRow = new List<List<Tile>>();
       //[row][col] or [Y][X]

              //For drawing tiles
              public static int tileLength = 0;
              public static int noOfTilesX = 20;       //Default noOfTilesXY
              public static int noOfTilesY = 20;




//SCREENS------------------------------------------------------------------------------
------------------
              //List of pbx screen names
              public static List<string> lstScreens = new List<string>();

              public static int maxNoOfTilesX = 0;
              public static int maxNoOfTilesY = 0;



//TILETYPES----------------------------------------------------------------------------
------------------
              //Keeps track of dicTileType keys (As Dictionaries are Hashtables)
              public static List<string> lstDicTileTypeKeys = new List<string>();
```

```
            //Holds info on each tileType
            public static Dictionary<string, Tuple<Color, int>> dicTileTypeInfo =
new Dictionary<string, Tuple<Color, int>>();
            //Holds list of tiles in map of certain tileTypes
            public static Dictionary<string, HashSet<string>> dicTileTypeTiles = new
Dictionary<string, HashSet<string>>();

            //Unset custom tileType panels in tabTiles
            public static bool mapReadyToEdit = true;



//MOUSE_HIGHLIGHTS--------------------------------------------------------------------
------------------
            //Holds dictionary key
            public static string selectedTileType = "";

            //For tile highlight
            public static int newTileHighlightX = 0;
            public static int newTileHighlightY = 0;
            public static int curTileHighlightX = 0;
            public static int curTileHighlightY = 0;



//PATHFINDING-------------------------------------------------------------------------
------------------
            public static Color pathColour = Color.DarkRed;
            public static bool readyToPathfind = true;
            public static bool donePathfinding = false;

            //For adding points to find shortest path between
            public static bool allowEditPoints = false;
            public static List<string> lstPointsToPathfind = new List<string>();

            public static bool fastSearch = true;

            //TEXT
OUTPUT------------------------------------------------------------------------------
------------
            public static int noOfIndents = 0;
            public static int indentFactor = 6;
        }

        public class Tile
        {
            //Name, TileType
            string tileCoords;
            string tileType;

            //Constructor
            public Tile(string _tileCoords, string _tileType)
```

```
        {
          tileCoords = _tileCoords;
          tileType = _tileType;
        }

        #region tileGetters
        public string getCoords()
        {
          return tileCoords;
        }

        public string getTileType()
        {
          return tileType;
        }

        //Refers to tileType dictionary
        public Color getColour()
        {
          Color tileColour = Global.dicTileTypeInfo[tileType].Item1;
          return tileColour;
        }

        public int getWeight()
        {
          int tileWeight = Global.dicTileTypeInfo[tileType].Item2;
          return tileWeight;
        }
        #endregion

        #region tileSetters
        public void setTileType(string _tileType)
        {
          tileType = _tileType;
        }
        #endregion

}

//For use in pathfinding algorithms that require sorting the openSet
public class PriorityListDijkstra
   {
        //Tentative distance, Coords
        List<Tuple<int, string>> lst;

        public PriorityListDijkstra()
        {
          lst = new List<Tuple<int, string>>();
        }

        public void Add(Tuple<int, string> input)
        {
          lst.Add(input);
          int indexNo = lst.Count - 1;
```

```
            bool loopEnd = false;

            //Sort
            while (loopEnd == false && indexNo > 0)
            {
                if (lst[indexNo].Item1 >= lst[indexNo - 1].Item1)
                {
                 //Swap
                 Tuple<int, string> temp = lst[indexNo - 1];
                 lst[indexNo - 1] = lst[indexNo];
                   lst[indexNo] = temp;

                 indexNo--;
                }
                else
                {
                 loopEnd = true;
                }
            }
        }

        public Tuple<int, string> dequeue()
        {
            Tuple<int, string> output = lst[lst.Count - 1];
            lst.RemoveAt(lst.Count - 1);
            return output;
        }

        public bool Contains(string coords)
    {
            bool contains = false;

            for (int i = 0; i < lst.Count(); i++)
            {
                if (lst[i].Item2 == coords)
                {
                 contains = true;
                 i = lst.Count() - 1;
                }
            }

            return contains;
        }

        public int Count()
        {
            return lst.Count();
        }

        public void Clear()
        {
            lst.Clear();
        }
}
```

```
public class PriorityListAStar
{
        //G-Score, F-Score, Coords
        List<Tuple<int, int, string>> lst;

        public PriorityListAStar()
        {
          lst = new List<Tuple<int, int, string>>();
        }


        public void Add(Tuple<int, int, string> input)
        {
          lst.Add(input);
          int indexNo = lst.Count - 1;
          bool loopEnd = false;

          //Sort by f-score
          while (loopEnd == false && indexNo > 0)
          {
              if (lst[indexNo].Item2 >= lst[indexNo - 1].Item2)
              {
               //Swap
                   Tuple<int, int, string> temp = lst[indexNo - 1];
               lst[indexNo - 1] = lst[indexNo];
               lst[indexNo] = temp;

               indexNo--;
              }
              else
               {
               loopEnd = true;
              }
          }
          loopEnd = false;
        }

        public Tuple<int, int, string> dequeue()
        {
          Tuple<int, int, string> output = lst[lst.Count - 1];
          lst.RemoveAt(lst.Count - 1);
          return output;
        }

        public bool Contains(string coords)
        {
          bool contains = false;

          for (int i = 0; i < lst.Count(); i++)
          {
              if (lst[i].Item3 == coords)
              {
               contains = true;
```

```
                    i = lst.Count() - 1;
                  }
              }

              return contains;
            }

            public int Count()
            {
              return lst.Count();
            }

            public void Clear()
            {
              lst.Clear();
            }
        }

#endregion
```

Initialisation

The code below deals with the start-up process of the program's controls:

```
#region Initialisation

        private void Form1_Load(object sender, EventArgs e)
        {
                this.FormBorderStyle = FormBorderStyle.None;
                this.WindowState = FormWindowState.Maximized;

             InitialiseTileTypes();
             InitialiseMap();

             InitialiseControls();
             AddHandlers();

                //Draws default grid
             OutputMap();
        }

        private void InitialiseTileTypes()
        {
                Global.lstDicTileTypeKeys.Add("Empty");
                Global.lstDicTileTypeKeys.Add("Wall");

                Global.dicTileTypeInfo.Add(Global.lstDicTileTypeKeys[0], new
Tuple<Color, int>(Color.White, 1));
                Global.dicTileTypeInfo.Add(Global.lstDicTileTypeKeys[1], new
Tuple<Color, int>(Color.Black, -1));

                Global.dicTileTypeTiles.Add(Global.lstDicTileTypeKeys[0], new
HashSet<string>());
                Global.dicTileTypeTiles.Add(Global.lstDicTileTypeKeys[1], new
HashSet<string>());
        }

        private void InitialiseMap()
        {
                //Default starting map vals
                Global.noOfTilesX = 20;
                Global.noOfTilesY = 20;

                //Fill one row at a time - Values are colours
                for (int i = 0; i < Global.noOfTilesY; i++)
                {
                  List<Tile> MapColumn = new List<Tile>();     //New column list for
each row
                  Tile newTile;

                  //Fill current row with columns
                  for (int j = 0; j < Global.noOfTilesX; j++)
```

```
                {
                    newTile = new Tile(j + "," + i, "Empty");
                    MapColumn.Add(newTile);
                    Global.dicTileTypeTiles["Empty"].Add(newTile.getCoords());
                }
                Global.lstMapRow.Add(MapColumn);
            }
        }

        #region initTabCtrl

        private void InitialiseControls()
        {
            pnlGrid.Location = new Point(30, 30);
            int pnlGridWidth = ((3 * (Screen.FromControl(this).WorkingArea.Width -
pnlGrid.Location.X)) / 4) + 1;
            int pnlGridHeight = Screen.FromControl(this).WorkingArea.Height -
pnlGrid.Location.Y + 1;
            pnlGrid.Size = new Size(pnlGridWidth, pnlGridHeight);

            tabCtrl.Location = new Point(pnlGrid.Location.X + pnlGridWidth + 30,
pnlGrid.Location.Y);
            tabCtrl.Size = new Size(Screen.FromControl(this).WorkingArea.Width -
(tabCtrl.Location.X + 30), pnlGridHeight - 40);

            //Start pathfinding button
            Button btnStartPathfind = new Button();
            btnStartPathfind.Name = "btnStartPathfind";
            btnStartPathfind.Location = new Point(tabCtrl.Location.X,
tabCtrl.Location.Y + tabCtrl.Height + 10);
            btnStartPathfind.Size = new Size(tabCtrl.Width, 30);
            btnStartPathfind.Text = "Start pathfinding";
            btnStartPathfind.Click += btnStartPathfind_Click;
            this.Controls.Add(btnStartPathfind);

            //tabScreen controls
            tabScreen.AutoScroll = true;
            initialiseTabScreen();

            //tabTiles controls
            tabTiles.AutoScroll = true;
            initialiseTabTiles();

            //tabOutput controls
            initialiseTabOutput();
        }

        private void addPnlAddSetCtrls(Panel pnl, string source)
        {
            //Button to add new tileTypes / Screens
            Button btnAdd = new Button();
            btnAdd.Font = new Font("Microsoft Sans Serif", 13.0f, FontStyle.Bold);
            btnAdd.Text = "+";
            btnAdd.Location = new Point(0, 0);
```

```
            btnAdd.Size = new Size(30, 30);

            pnl.Controls.Add(btnAdd);

              if (source == "SCREENS")
              {
                btnAdd.Name = "btnAddScreen";
                btnAdd.Click += btnAddScreen_Click;
              }
              else if (source == "TILETYPES")
              {
                //Button to set custom tileType options - Will only have one
(initially invisible)
                Button btnSet = new Button();
                btnSet.Name = "btnSetTileTypes";       //Needed so we can refer to it
when making it visible/invisible
                btnSet.Text = "Set";
                btnSet.Size = new Size(50, 30);
                btnSet.Location = new Point(226 - btnSet.Width, 0);         //Line up
end of btn with end of textbox

                pnl.Controls.Add(btnSet);

                btnAdd.Name = "btnAddTileType";

                btnSet.Visible = false;         //Visible only when at least one
custom tileType exists

                btnAdd.Click += btnAddTileType_Click;
                btnSet.Click += btnSetTileTypes_Click;
              }
       }

        #region initTabScreen
       private void initialiseTabScreen()
       {
              //Panel to set noOfTilesXY
              Panel pnlScreenSize = new Panel();
            initPnlScreenSize(pnlScreenSize);

              //Clear map button
              Button btnClearMap = new Button();
            btnClearMap.Text = "Clear Map";
            btnClearMap.AutoSize = true;
            btnClearMap.Location = new Point(tabCtrl.Width - btnClearMap.Width - 30,
pnlScreenSize.Location.Y);
            tabScreen.Controls.Add(btnClearMap);
            btnClearMap.Click += btnClearMap_Click;

              //Maximise number of tiles button
              Button btnMaximise = new Button();
            btnMaximise.Text = "Maximise";
            btnMaximise.AutoSize = true;
```

```
            btnMaximise.Location = new Point(btnClearMap.Location.X,
btnClearMap.Location.Y + 30);
            tabScreen.Controls.Add(btnMaximise);
            btnMaximise.Click += btnMaximiseNoOfTiles_Click;

              //For adding nodes to find shortest path between
              Panel pnlAddPoints = new Panel();
            pnlAddPoints.Name = "pnlAddPoints";
             pnlAddPoints.Location = new Point(pnlScreenSize.Location.X,
pnlScreenSize.Location.Y + pnlScreenSize.Height + 10);
            initPnlAddPathPoints(pnlAddPoints);

              //Line to separate pnlScreenSize and Screen Options
              Label lblSeparator = new Label();
            lblSeparator.Name = "lblSeparator";
            lblSeparator.Location = new Point(30, pnlAddPoints.Location.Y +
pnlAddPoints.Height);
              string strDashes = "";
              for (int i = 0; i < (tabCtrl.Width - 120) / 3; i++)
              {
                strDashes += "-";
              }
            lblSeparator.Text = strDashes;
            lblSeparator.AutoSize = true;
            tabScreen.Controls.Add(lblSeparator);

              //First screen - Default
              Panel pnlDefScreen = new Panel();
            pnlDefScreen.Name = "Screen0";
            pnlDefScreen.Location = new Point(30, lblSeparator.Location.Y +
lblSeparator.Height + 10);
            pnlDefScreen.Size = new Size(tabCtrl.Width - 60,
Global.screenOptionsHeight);
            pnlDefScreen.BackColor = Color.FromArgb(70, Color.LawnGreen);

              //NOTE: For all screens, since all options are selected via combobox,
there is no need to set algorithms
              addScreenPanelCtrls(pnlDefScreen);
            tabScreen.Controls.Add(pnlDefScreen);

              //Controls for adding/setting new tile types
              Panel pnlAddSet = new Panel();
            pnlAddSet.Name = "pnlAddSet";
            pnlAddSet.Location = new Point(30, pnlDefScreen.Location.Y +
pnlDefScreen.Height);
            pnlAddSet.Size = new Size(tabCtrl.Width - 60, 30);
            addPnlAddSetCtrls(pnlAddSet, "SCREENS");
            tabScreen.Controls.Add(pnlAddSet);
      }

      private void initPnlScreenSize(Panel pnlScreenSize)
      {
            pnlScreenSize.Name = "pnlScreenSize";
            pnlScreenSize.Location = new Point(10, 10);
```

```
            pnlScreenSize.Size = new Size(200, 110);

              Label lblWidth = new Label();
            lblWidth.Text = "Width:";
            lblWidth.Location = new Point(20, 20);
            lblWidth.AutoSize = true;
            pnlScreenSize.Controls.Add(lblWidth);

              TextBox tbxWidth = new TextBox();
            tbxWidth.Name = "tbxWidth";
            tbxWidth.Text = Global.noOfTilesX.ToString();
            tbxWidth.Location = new Point(70, lblWidth.Location.Y - 2);
            pnlScreenSize.Controls.Add(tbxWidth);

              Label lblHeight = new Label();
            lblHeight.Text = "Height:";
            lblHeight.Location = new Point(20, lblWidth.Location.Y + 30);
            lblHeight.AutoSize = true;
            pnlScreenSize.Controls.Add(lblHeight);

              TextBox tbxHeight = new TextBox();
            tbxHeight.Name = "tbxHeight";
            tbxHeight.Text = Global.noOfTilesY.ToString();
            tbxHeight.Location = new Point(70, lblHeight.Location.Y - 2);
            pnlScreenSize.Controls.Add(tbxHeight);

              Button btnSetNoOfTiles = new Button();
            btnSetNoOfTiles.Text = "Set";
            btnSetNoOfTiles.Location = new Point(tbxHeight.Location.X,
tbxHeight.Location.Y + 30);
            btnSetNoOfTiles.Size = new Size(tbxHeight.Width, 25);
            pnlScreenSize.Controls.Add(btnSetNoOfTiles);
            btnSetNoOfTiles.Click += btnSetNoOfTiles_Click;

            tabScreen.Controls.Add(pnlScreenSize);
        }

        private void initPnlAddPathPoints(Panel pnlAddPoints)
        {
                pnlAddPoints.Size = new Size(300, 60);

              Label lblAddPoints = new Label();
            lblAddPoints.Name = "lblAddPoints";
            lblAddPoints.Location = new Point(20, 0);
            lblAddPoints.Text = "Path points edit mode: OFF";
            lblAddPoints.AutoSize = true;
            pnlAddPoints.Controls.Add(lblAddPoints);

              Button btnAddPoints = new Button();
            btnAddPoints.Text = "Toggle add points";
            btnAddPoints.AutoSize = true;
            btnAddPoints.Location = new Point(20, 20);
            pnlAddPoints.Controls.Add(btnAddPoints);
            btnAddPoints.Click += btnAddPathPoints_Click;
```

```
            tabScreen.Controls.Add(pnlAddPoints);
        }

        private void addAlgorithmsToCmbBox(ComboBox cmb)
        {
            cmb.Items.Add("BFS");
            cmb.Items.Add("Dijkstra");
            cmb.Items.Add("A*");

            cmb.Text = "BFS";      //Default
        }

         #endregion

         #region initTabTiles
        private void initialiseTabTiles()
        {
               /*NOTES:
               * Each tile topLeft 70 px. apart in y (= customPanelHeight)
               * Each tile size (51, 51) px.
               * Labels and tile topLeft 70 px. apart in x
               * Label and textbox 6 spaces apart in x (for 'Name'): '       '
               * Name and weight label topLeft 25 px. apart in y
               * Panel's x ends at (-120 from tabCtrl.Width) +30 Location.X
               * TopLefts: pnlDefTileTypes = 30 px.
               *           lblSeparator = 169 px.
               *                    pnlAddSet = 199 px.
               *
               *POSSIBILE IMPROVEMENTS:
               * Have locations and sizes of controls in tabTiles as variables with
default vals, but allow user to customise (MINIMUM SCREEN RES?)
               */

               //Default tile types
              addDefTileTypes();

               //Line to separate default and custom tileTypes
               Label lblSeparator = new Label();
              lblSeparator.Name = "lblSeparator";
              lblSeparator.Location = new Point(30, 168);
               string strDashes = "";
               for (int i = 0; i < (tabCtrl.Width - 120) / 3; i++)
               {
                 strDashes += "-";
               }
              lblSeparator.Text = strDashes;
              lblSeparator.AutoSize = true;
              tabTiles.Controls.Add(lblSeparator);

               //Controls for adding/setting new tile types
               Panel pnlAddSet = new Panel();
              pnlAddSet.Name = "pnlAddSet";
              pnlAddSet.Location = new Point(30, lblSeparator.Location.Y + 30);
```

```
            pnlAddSet.Size = new Size(tabCtrl.Width - 60, 30);
            addPnlAddSetCtrls(pnlAddSet, "TILETYPES");
            tabTiles.Controls.Add(pnlAddSet);
        }

        private void addDefTileTypes()
        {
              //pnlEmpty
              Panel pnlEmpty = new Panel();
            pnlEmpty.Name = "Empty";
            pnlEmpty.Location = new Point(30, 30);
            pnlEmpty.Size = new Size(tabCtrl.Width - 60, 70);

              PictureBox pbxEmpty = new PictureBox();
            pbxEmpty.Name = "pbxEmpty";
            pbxEmpty.Size = new Size(51, 51);
            pbxEmpty.Location = new Point(0, 0);
            drawTileTypePbx(pbxEmpty, Global.dicTileTypeInfo["Empty"].Item1);
            pbxEmpty.MouseClick += tileTypeColour_Click;      //Move tileType selector
on pbx click
            pnlEmpty.Controls.Add(pbxEmpty);

              Label lblEmptyName = new Label();
            lblEmptyName.AutoSize = true;
            lblEmptyName.Text = "Name:    Empty";
            lblEmptyName.Location = new Point(70, pbxEmpty.Location.Y);
            pnlEmpty.Controls.Add(lblEmptyName);

              Label lblEmptyWeight = new Label();
            lblEmptyWeight.AutoSize = true;
            lblEmptyWeight.Text = "Weight:      1";
            lblEmptyWeight.Location = new Point(70, pbxEmpty.Location.Y + 25);
            pnlEmpty.Controls.Add(lblEmptyWeight);

            tabTiles.Controls.Add(pnlEmpty);

              //pnlWall
              Panel pnlWall = new Panel();
            pnlWall.Name = "Wall";
            pnlWall.Location = new Point(30, 100);
            pnlWall.Size = new Size(tabCtrl.Width - 60, 70);

              PictureBox pbxWall = new PictureBox();
            pbxWall.Name = "pbxWall";
            pbxWall.Size = new Size(51, 51);
            pbxWall.Location = new Point(0, 0);
              drawTileTypePbx(pbxWall, Color.Black);
            pbxWall.MouseClick += tileTypeColour_Click;       //Move tileType selector
on pbx click
            pnlWall.Controls.Add(pbxWall);

              Label lblWallName = new Label();
            lblWallName.AutoSize = true;
            lblWallName.Text = "Name:    Wall";
```

```
            lblWallName.Location = new Point(70, pbxWall.Location.Y);
            pnlWall.Controls.Add(lblWallName);

              Label lblWallWeight = new Label();
            lblWallWeight.AutoSize = true;
            lblWallWeight.Text = "Weight: -1";
            lblWallWeight.Location = new Point(70, pbxWall.Location.Y + 25);
            pnlWall.Controls.Add(lblWallWeight);

            tabTiles.Controls.Add(pnlWall);

            initTileTypeSelector(pnlWall);
        }

        private void initTileTypeSelector(Panel pnlDefTileType)
        {
              PictureBox pbxTileSelect = new PictureBox();
            pbxTileSelect.Name = "pbxTileSelect";
            pbxTileSelect.BackColor = Color.Red;
            pbxTileSelect.Size = new Size(10, 10);
            pbxTileSelect.Location = new Point(pnlDefTileType.Location.X - 22,
pnlDefTileType.Location.Y + 20);

              Global.selectedTileType = pnlDefTileType.Name;

            tabTiles.Controls.Add(pbxTileSelect);
        }

        private void drawTileTypePbx(PictureBox pbx, Color fillClr)
        {
              Bitmap bmp = new Bitmap(pbx.Width, pbx.Height);
              Graphics gfx = Graphics.FromImage(bmp);

            gfx.FillRectangle(new SolidBrush(fillClr), 0, 0, bmp.Width - 1, bmp.Height
- 1);
            gfx.DrawRectangle(Pens.Black, 0, 0, bmp.Width - 1, bmp.Height - 1);
            pbx.Image = bmp;
        }

         #endregion

         #region initTabOutput
        private void initialiseTabOutput()
        {
              ListBox lstOutput = new ListBox();
            lstOutput.Enabled = false;
            lstOutput.Location = new Point(5, 5);
            lstOutput.Size = new Size(tabCtrl.Width - 20, tabCtrl.Height - 36);
            tabOutput.Controls.Add(lstOutput);
        }

        private void outputToTbxOutput(string output)
        {
              ListBox lstOutput = (ListBox)tabOutput.Controls[0];
```

```
            bool addExtraLine = false;

            if (output.StartsWith("..."))
            {
              Global.noOfIndents--;
              addExtraLine = (Global.noOfIndents == 0) ? true : false;
            }

            //Add indents to output
            int noOfSpaces = Global.indentFactor * Global.noOfIndents;
            string spaces = "";
            for (int i = 0; i < noOfSpaces; i++)
            {
              spaces += " ";
            }

            //Add items to listBox
          lstOutput.Items.Add(spaces + output);
            if (addExtraLine) { lstOutput.Items.Add(Environment.NewLine); }

            if (output.EndsWith("..."))
            {
              Global.noOfIndents++;
            }

            //Remove excess lines - Prevents vertical scrollbar from appearing
            while (lstOutput.Items.Count * lstOutput.ItemHeight > lstOutput.Height)
            {
              lstOutput.Items.RemoveAt(0);
            }
        }
         #endregion

         #endregion

        private void AddHandlers()
        {
               //Menu Bar - NEW
          menuBarNew.Click += menuBarNew_Click;
          menuBarImportImage.Click += menuBarImportImage_Click;
          menuBarExportBmp.Click += menuBarExportBmp_Click;
          menuBarExit.Click += menuBarExit_Click;

            //Menu Bar - EDIT
          menuBarClearMap.Click += btnClearMap_Click;
        }

#endregion
```

tabCtrlMethods

The code below deals with user interactions with the tabs and its components:

```
#region tabCtrlMethods

        #region tabScreenMethods
       private void btnSetNoOfTiles_Click(object sender, EventArgs e)
       {
              Button btnSet = (Button)sender;
              Panel pnlScreenSize = (Panel)btnSet.Parent;
              TextBox tbxWidth = (TextBox)pnlScreenSize.Controls.Find("tbxWidth",
false).FirstOrDefault();
              TextBox tbxHeight = (TextBox)pnlScreenSize.Controls.Find("tbxHeight",
false).FirstOrDefault();

              int noOfTilesX, noOfTilesY;

              string tbxOutput = string.Format("Attempting to change map size to {0} x
{1}...", tbxWidth.Text, tbxHeight.Text);
            outputToTbxOutput(tbxOutput);

              //Checks if user inputs are integers first
              if (int.TryParse(tbxWidth.Text, out noOfTilesX))
              {
                if (int.TryParse(tbxHeight.Text, out noOfTilesY))
                {
                    noOfTilesValidation(noOfTilesX, noOfTilesY);
                }
                else
                {
                    tbxOutput = string.Format("Height '{0}' is not an integer",
tbxHeight.Text);
                    outputToTbxOutput(tbxOutput);
                    tbxOutput = string.Format("...map size of {0} x {1} rejected",
tbxWidth.Text, tbxHeight.Text);
                    outputToTbxOutput(tbxOutput);
                    MessageBox.Show("Invalid height");
                }
              }
              else
              {
                tbxOutput = string.Format("Width '{0}' is not an integer",
tbxWidth.Text);
                outputToTbxOutput(tbxOutput);
                tbxOutput = string.Format("...map size of {0} x {1} rejected",
tbxWidth.Text, tbxHeight.Text);
                outputToTbxOutput(tbxOutput);
                MessageBox.Show("Invalid width");
              }
       }
```

```
        private void noOfTilesValidation(int noOfTilesX, int noOfTilesY)
        {
                string tbxOutput = "";

                if (noOfTilesX != 0 && noOfTilesY != 0)
                {
                  int oldNoOfTilesX = Global.noOfTilesX;
                  int oldNoOfTilesY = Global.noOfTilesY;

                  //Update current noOfTiles on map for trial fit
                  Global.noOfTilesX = noOfTilesX;
                  Global.noOfTilesY = noOfTilesY;
                  Tuple<int, string> tileLengthStatus = trialFitScreen();
                   int testTileLength = tileLengthStatus.Item1;
                  string lastAction = tileLengthStatus.Item2;

                  //Note: Revert must be made for 'RebuildMap' to work correctly
                  Global.noOfTilesX = oldNoOfTilesX;
                Global.noOfTilesY = oldNoOfTilesY;

                  if (testTileLength == -1)
                  {
                      //Inform user what caused error by testing whether X/Y/Both caused
tileLength to go below min
                      if (Global.lstScreens.Count == 1)
                      {
                       if (lastAction == "X")
                       {
                              tbxOutput = string.Format("Width '{0}' too large",
noOfTilesX);
                              outputToTbxOutput(tbxOutput);
                              MessageBox.Show("Width too large");
                       }
                       else if (lastAction == "Y")
                       {
                              tbxOutput = string.Format("Width '{0}' too large",
noOfTilesY);
                              outputToTbxOutput(tbxOutput);
                              MessageBox.Show("Height too large");
                       }
                       else if (lastAction == "XY")
                       {
                              tbxOutput = string.Format("Both width '{0}' and height
'{1}' too large", noOfTilesX, noOfTilesY);
                              outputToTbxOutput(tbxOutput);
                              MessageBox.Show("Both width and height too large");
                       }
                      }
                      else
                      {
                       //Note: For multiple screens, it's too difficult to tell whether
it's width/height/both that caused tileLength to go below minimum
                       tbxOutput = string.Format("Both width '{0}' and height '{1}' too
large", noOfTilesX, noOfTilesY);
```

```
                    outputToTbxOutput(tbxOutput);
                MessageBox.Show("Unable to change number of tiles");
                }

                tbxOutput = string.Format("...map size of {0} x {1} rejected",
noOfTilesX, noOfTilesY);
                outputToTbxOutput(tbxOutput);
            }
            else
            {
                tbxOutput = string.Format("...map size of {0} x {1} accepted",
noOfTilesX, noOfTilesY);
                outputToTbxOutput(tbxOutput);

                //Passed fitting
                rebuildMap(noOfTilesX, noOfTilesY);

                //Update current noOfTiles on map
                Global.noOfTilesX = noOfTilesX;
                Global.noOfTilesY = noOfTilesY;

                //Accept testTileLength
                Global.tileLength = testTileLength;

                resetPathfinding();
                placeScreens();
                DrawFullGrid();
            }
        }
        else
        {
            tbxOutput = string.Format("Can't have width/height of 0");
            outputToTbxOutput(tbxOutput);
            tbxOutput = string.Format("...map size of {0} x {1} rejected",
noOfTilesX, noOfTilesY);
            outputToTbxOutput(tbxOutput);
            MessageBox.Show("Width/Height cannot be 0");
        }
    }

    private void rebuildMap(int noOfColumns, int noOfRows)
    {
        //MAP REBUILD PSEUDOCODE - Row = Y, Column = X
        /* If more rows than original
        *      For each new row
        *          Add new no. of columns (tiles)
        *          Add each new tile to set of tiles of some tileType
        * Else if less rows than original
        *      For each extra row
        *          Delete each tile of this row from dicTileTypeTiles
        *          Delete extra row
        * End If
        *
        * If more columns than original
```

```
            *      Add new columns (tiles) to original rows that are still within
map
            *      Add each new tile to set of tiles of some tileType
            * Else if less columns than original
            *      For each original row that are still within map
            *             Delete extra columns (tiles)
            *             Delete each of these tiles from dicTileTypeTiles
            */

            string tbxOutput = "";
            tbxOutput = string.Format("Rebuilding map from {0} x {1} to {2} x {3}...",
Global.noOfTilesX, Global.noOfTilesY, noOfColumns, noOfRows);
            outputToTbxOutput(tbxOutput);

            //Needed for columns if rows deleted, not all original rows will still
be on map
            int originalRows = Global.noOfTilesY;

            //If more rows than original
            if (noOfRows > Global.noOfTilesY)
            {
              tbxOutput = string.Format("Too little rows, adding more");
              outputToTbxOutput(tbxOutput);
              addMapRows(noOfColumns, noOfRows);
            }
          //If less rows than original
            else if (noOfRows < Global.noOfTilesY)
            {
              tbxOutput = string.Format("Too many rows, removing more");
              outputToTbxOutput(tbxOutput);
              deleteMapRows(noOfColumns, noOfRows, ref originalRows);
            }

            //If more columns than original
            if (noOfColumns > Global.noOfTilesX)
            {
              tbxOutput = string.Format("Too little columns, adding more");
              outputToTbxOutput(tbxOutput);
              addMapColumns(noOfColumns, noOfRows, originalRows);
            }
            //If less columns than original
            else if (noOfColumns < Global.noOfTilesX)
            {
              tbxOutput = string.Format("Too many columns, removing some");
              outputToTbxOutput(tbxOutput);
              deleteMapColumns(noOfColumns, noOfRows, originalRows);
            }

            tbxOutput = string.Format("...done rebuilding map");
            outputToTbxOutput(tbxOutput);
        }

        private void addMapRows(int noOfColumns, int noOfRows)
        {
```

```
            //Add new rows with new no. of columns
            for (int row = Global.noOfTilesY; row < noOfRows; row++)
            {
              List<Tile> MapColumn = new List<Tile>();
              Tile newTile;

              for (int col = 0; col < noOfColumns; col++)
              {
                  newTile = new Tile(col + "," + row, "Empty");
                  MapColumn.Add(newTile);
                  Global.dicTileTypeTiles["Empty"].Add(newTile.getCoords());
              }

              Global.lstMapRow.Add(MapColumn);
            }
        }

        private void deleteMapRows(int noOfColumns, int noOfRows, ref int originalRows)
        {
            Tile tileDelete;

            //Delete extra rows
            for (int row = Global.noOfTilesY - 1; row >= noOfRows; row--)
            {
              //Delete tiles from dictTileTypeTiles
              for (int col = 0; col < Global.noOfTilesX; col++)
              {
                  tileDelete = Global.lstMapRow[row][col];

Global.dicTileTypeTiles[tileDelete.getTileType()].Remove(tileDelete.getCoords());
              }

              Global.lstMapRow.RemoveAt(row);
            }

            bool loopFinish = false;
            int coordsIndex = 0;

            //If there are points to pathfind
            if (Global.lstPointsToPathfind.Count() > 0)
            {
              //Delete path points that sat on deleted rows
              while (loopFinish == false)
              {
                  int row =
int.Parse(Global.lstPointsToPathfind[coordsIndex].Split(',')[1]);
                  if (row > noOfRows - 1)
                  {
                   reorderStringList(ref Global.lstPointsToPathfind, coordsIndex);
                  }
                  else
                  {
                   coordsIndex++;
                  }
```

```
                    //If next index is beyond list bounds
                    if (coordsIndex == Global.lstPointsToPathfind.Count())
                    {
                     loopFinish = true;
                    }
                }
            }

             //Update original rows to original rows that still exit on map
           originalRows = noOfRows;
        }

        private void addMapColumns(int noOfColumns, int noOfRows, int originalRows)
        {
             //Add new columns to original rows
             for (int row = 0; row < originalRows; row++)
             {
               List<Tile> MapColumn = Global.lstMapRow[row];
               Tile newTile;

               for (int col = Global.noOfTilesX; col < noOfColumns; col++)
               {
                   newTile = new Tile(col + "," + row, "Empty");
                   MapColumn.Add(newTile);
                   Global.dicTileTypeTiles["Empty"].Add(newTile.getCoords());
               }
             }
        }

        private void deleteMapColumns(int noOfColumns, int noOfRows, int originalRows)
        {
             //Delete extra columns for each original row
             for (int row = 0; row < originalRows; row++)
             {
               List<Tile> MapColumn = Global.lstMapRow[row];
               Tile tileDelete;

               for (int col = Global.noOfTilesX - 1; col >= noOfColumns; col--)
               {
                   tileDelete = Global.lstMapRow[row][col];
                   MapColumn.RemoveAt(col);

Global.dicTileTypeTiles[tileDelete.getTileType()].Remove(tileDelete.getCoords());
               }
             }

             //If there are points to pathfind
             if (Global.lstPointsToPathfind.Count() > 0)
             {
               bool loopFinish = false;
               int coordsIndex = 0;

               //Delete path points that sat on deleted columns
```

```
                while (loopFinish == false)
                {
                 int col =
int.Parse(Global.lstPointsToPathfind[coordsIndex].Split(',')[0]);
                    if (col > noOfColumns - 1)
                    {
                     reorderStringList(ref Global.lstPointsToPathfind, coordsIndex);
                    }
                    else
                    {
                     coordsIndex++;
                    }

                    //If next index is beyond list bounds
                    if (coordsIndex == Global.lstPointsToPathfind.Count())
                    {
                     loopFinish = true;
                    }
                }
              }
       }

       private void btnClearMap_Click(object sender, EventArgs e)
       {
              DialogResult newMapResult = MessageBox.Show("Clear the map?", "Clear
Map", MessageBoxButtons.YesNo, MessageBoxIcon.Question);
              if (newMapResult == DialogResult.Yes)
              {
                string tbxOutput = "";

                tbxOutput = string.Format("Clearing map...");
                outputToTbxOutput(tbxOutput);

                //Holds coords to update on grid
                HashSet<string> hsetTilesToUpdate = new HashSet<string>();

                //Transfer coords from non-empty tiles to empty tile, create copy of
transferred coords to hsetCoordsToUpdate, clear non-empty tile HashSet
                foreach (string tileType in Global.lstDicTileTypeKeys)
                {
                    if (tileType != "Empty")
                    {
                     foreach (string coords in Global.dicTileTypeTiles[tileType])
                     {
                            Tile tileToEdit = coordsToTile(coords);
                            tileToEdit.setTileType("Empty");

                            Global.dicTileTypeTiles["Empty"].Add(coords);
                            hsetTilesToUpdate.Add(coords);
                     }
                     Global.dicTileTypeTiles[tileType].Clear();
                    }
                }
                tbxOutput = string.Format("Set all tileTypes on map to \"Empty\"");
```

```
                outputToTbxOutput(tbxOutput);

                resetPathfinding();
                clearAllPathPoints();
                   updateTilesOnMap(hsetTilesToUpdate);

                //Reset overlays
                foreach (Panel pnlScreen in pnlGrid.Controls)
                {
                    PictureBox pbxGrid = (PictureBox)pnlScreen.Controls[0];
                    initPbxGridOverlay(pbxGrid);
                }

                tbxOutput = string.Format("Reset all overlays of all screens");
                outputToTbxOutput(tbxOutput);

                tbxOutput = string.Format("...map cleared");
                outputToTbxOutput(tbxOutput);
            }
        }

        private void btnMaximiseNoOfTiles_Click(object sender, EventArgs e)
        {
            string tbxOutput = "";

            tbxOutput = string.Format("Maximising map size for {0} screens...",
Global.lstScreens.Count());
            outputToTbxOutput(tbxOutput);

            rebuildMap(Global.maxNoOfTilesX, Global.maxNoOfTilesY);

              Global.noOfTilesX = Global.maxNoOfTilesX;
              Global.noOfTilesY = Global.maxNoOfTilesY;

              Global.tileLength = Global.minTileLength;

              //Change width and height textbox text
              Panel pnlScreenSize = (Panel)tabScreen.Controls.Find("pnlScreenSize",
false).FirstOrDefault();
              TextBox tbxWidth = (TextBox)pnlScreenSize.Controls.Find("tbxWidth",
false).FirstOrDefault();
              TextBox tbxHeight = (TextBox)pnlScreenSize.Controls.Find("tbxHeight",
false).FirstOrDefault();

            tbxWidth.Text = Global.maxNoOfTilesX.ToString();
            tbxHeight.Text = Global.maxNoOfTilesY.ToString();

            resetPathfinding();
            placeScreens();
            DrawFullGrid();

            tbxOutput = string.Format("...map size maximised");
            outputToTbxOutput(tbxOutput);
        }
```

```
        private void btnAddPathPoints_Click(object sender, EventArgs e)
        {
                if (!Global.donePathfinding)
                {
                  Button btnAddPoints = (Button)sender;
                  Panel pnlAddPoints = (Panel)btnAddPoints.Parent;
                  Label lblAddPoints = (Label)pnlAddPoints.Controls.Find("lblAddPoints",
false).FirstOrDefault();

                  string tbxOutput = "";

                  //Pathfinding point placement mode indicator
                  if (lblAddPoints.ForeColor != Color.Red)
                  {
                      lblAddPoints.Text = "Path points edit mode: ON";
                      lblAddPoints.ForeColor = Color.Red;
                      Global.allowEditPoints = true;

                      tbxOutput = string.Format("Path points edit mode set to ON");
                      outputToTbxOutput(tbxOutput);
                  }
                  else
                  {
                      lblAddPoints.Text = "Path points edit mode: OFF";
                      lblAddPoints.ForeColor = Color.Black;
                      Global.allowEditPoints = false;

                      tbxOutput = string.Format("Path points edit mode set to OFF");
                      outputToTbxOutput(tbxOutput);
                  }
                }
        }

        private void btnAddScreen_Click(object sender, EventArgs e)
        {
                /*ADDING PROCEDURE:
                * -Create new screen and show on pnlGrid
                * -Create new panel for screen list
                * -Base new panel name off new screen name (Just hold list of names with
a string list)
                * -Add controls to this panel
                */

                string tbxOutput = "";

               tbxOutput = string.Format("Attempting to add new screen...");
               outputToTbxOutput(tbxOutput);

                //Required for trialFitScreen - Reverted later on
                Global.lstScreens.Add("TEMP");

                Tuple<int, string> tileLengthStatus = trialFitScreen();
                int testTileLength = tileLengthStatus.Item1;
```

```
            Global.lstScreens.RemoveAt(Global.lstScreens.Count - 1);

            if (testTileLength != -1)
            {
              Button btnAdd = (Button)sender;
              Panel pnlAddSet = (Panel)btnAdd.Parent;

            Panel pnlScreenTab = new Panel();
              pnlScreenTab.Location = new Point(30, pnlAddSet.Location.Y);
              pnlScreenTab.Size = new Size(tabCtrl.Width - 60,
Global.screenOptionsHeight);
              pnlScreenTab.BackColor = Color.FromArgb(70, Color.LawnGreen);
              addScreenPanelCtrls(pnlScreenTab);
              tabScreen.Controls.Add(pnlScreenTab);

              //Moves add/set panel down
              pnlAddSet.Location = new Point(30, pnlAddSet.Location.Y +
pnlScreenTab.Height);

              //Hide add button if max no. of screens reached
              if (Global.lstScreens.Count() == Global.maxScreens)
              {
                  tbxOutput = string.Format("Max no. of screens reached");
                  outputToTbxOutput(tbxOutput);

                  btnAdd.Visible = false;
              }

              Global.tileLength = testTileLength;

              resetPathfinding();
              placeScreens();
              DrawFullGrid();

              tbxOutput = string.Format("...new screen successfully added");
              outputToTbxOutput(tbxOutput);
            }
            else
            {
              tbxOutput = string.Format("...could not add new screen");
              outputToTbxOutput(tbxOutput);

              MessageBox.Show("Unable to add new screen");
            }
      }

      private void addScreenPanelCtrls(Panel pnl)
      {
            pnl.Name = "Screen" + Global.lstScreens.Count().ToString();

             Label lblScreenName = new Label();
            lblScreenName.Name = "lblScreenName";
            lblScreenName.Text = pnl.Name;
```

```
            lblScreenName.Font = new Font("Microsoft Sans Serif", 11.0f);
          lblScreenName.AutoSize = true;
            lblScreenName.Location = new Point(0, 0);
            lblScreenName.BackColor = Color.Transparent;
            pnl.Controls.Add(lblScreenName);

              Global.lstScreens.Add(lblScreenName.Text);

          Label lblAlgorithm = new Label();
            lblAlgorithm.Text = "Algorithm:";
            lblAlgorithm.AutoSize = true;
            lblAlgorithm.Location = new Point(30, lblScreenName.Height + 5);
            lblAlgorithm.BackColor = Color.Transparent;
            pnl.Controls.Add(lblAlgorithm);

              ComboBox cmbAlgorithm = new ComboBox();
            cmbAlgorithm.Name = "cmbAlgorithm";
            addAlgorithmsToCmbBox(cmbAlgorithm);
            cmbAlgorithm.AutoSize = true;
             cmbAlgorithm.Location = new Point(lblAlgorithm.Location.X +
lblAlgorithm.Width + 10, lblAlgorithm.Location.Y);
            pnl.Controls.Add(cmbAlgorithm);
            cmbAlgorithm.SelectedIndexChanged += cmbAlgorithmChange;        //Handles
changes to algorithm selection

              //If not default screen
              if (Global.lstScreens.Count() > 1)
              {
                //Delete button
                Button btnDeleteScreen = new Button();
                btnDeleteScreen.Font = new Font("Microsoft Sans Serif", 13.0f,
FontStyle.Bold);
                btnDeleteScreen.Text = "x";
                btnDeleteScreen.Size = new Size(30, 30);
                btnDeleteScreen.Location = new Point(pnl.Width -
btnDeleteScreen.Width, 0);
                btnDeleteScreen.BackColor = Color.Transparent;
                pnl.Controls.Add(btnDeleteScreen);
                btnDeleteScreen.Click += btnDeleteScreen_Click;
              }
        }

        private void btnDeleteScreen_Click(object sender, EventArgs e)
        {
              /*DELETION PROCEDURE:
              * -Delete panel from pnlGrid
              * -Delete panel from screen list, keeping track of index to delete
              * -Reorganise panels screen
              */
              Button btnDelete = (Button)sender;
              Panel pnlToDelete = (Panel)btnDelete.Parent;
              Panel pnlAddSet = (Panel)tabScreen.Controls.Find("pnlAddSet",
false).FirstOrDefault();
```

```
            string tbxOutput = "";
            tbxOutput = string.Format("Deleting {0}...", pnlToDelete.Name);
            outputToTbxOutput(tbxOutput);

            //Make btnAdd in pnlAddSet to be visible again
            Button btnAdd = (Button)pnlAddSet.Controls.Find("btnAddScreen",
false).FirstOrDefault();
            btnAdd.Visible = true;

            //"Screen" = 6 char
            int indexToDelete = int.Parse(pnlToDelete.Name.Substring(6,
pnlToDelete.Name.Count() - 6));

            //Moves add/set panel up
            pnlAddSet.Location = new Point(30, pnlAddSet.Location.Y -
pnlToDelete.Height);

            //Delete panel
            tabScreen.Controls.Remove(pnlToDelete);

            //Reorganise panel and screen list
            for (int i = indexToDelete; i < Global.lstScreens.Count() - 1; i++)
        {
                //Moves next panel up one slot
                Panel pnlToMove = (Panel)tabScreen.Controls.Find(Global.lstScreens[i +
1], false).FirstOrDefault();
                pnlToMove.Location = new Point(pnlToMove.Location.X,
pnlToMove.Location.Y - pnlToMove.Height);

                //Renames screen based on new position
                string newScreenName = "Screen" + i.ToString();
                pnlToMove.Name = newScreenName;

                Label lblScreenName = (Label)pnlToMove.Controls.Find("lblScreenName",
false).FirstOrDefault();
                lblScreenName.Text = newScreenName;

                Global.lstScreens[i] = newScreenName;
            }

            tbxOutput = string.Format("Screens reorganised");
            outputToTbxOutput(tbxOutput);

            //Remove last element in screen list
            Global.lstScreens.RemoveAt(Global.lstScreens.Count() - 1);

            //Refreshes pnlGrid and resets pathfinding
            OutputMap();
            resetPathfinding();

            tbxOutput = string.Format("...screen{0} deleted", indexToDelete);
            outputToTbxOutput(tbxOutput);
        }
```

```
        private void cmbAlgorithmChange(object sender, EventArgs e)
        {
             ComboBox cmbAlgorithm = (ComboBox)sender;
               Panel pnlScreenOptions = (Panel)cmbAlgorithm.Parent;

               //If A*, display heuristic options
               if (cmbAlgorithm.Text == "A*")
               {
                 addHeuristicsOptions(pnlScreenOptions, cmbAlgorithm);
               }
               else
               {
                 //Delete heuristic options, if available
                 if (pnlScreenOptions.Controls.Find("cmbHeuristic",
false).FirstOrDefault() != null)
                 {
                     pnlScreenOptions.Controls.RemoveByKey("lblHeuristic");
                     pnlScreenOptions.Controls.RemoveByKey("cmbHeuristic");
                 }
               }
        }

        private void addHeuristicsOptions(Panel pnlScreenOptions, ComboBox
cmbAlgorithm)
        {
               Label lblHeuristic = new Label();
             lblHeuristic.Name = "lblHeuristic";
             lblHeuristic.Text = "Heuristic:";
             lblHeuristic.AutoSize = true;
             lblHeuristic.Location = new Point(30, cmbAlgorithm.Location.Y + 30);
             lblHeuristic.BackColor = Color.Transparent;
             pnlScreenOptions.Controls.Add(lblHeuristic);

               ComboBox cmbHeuristic = new ComboBox();
             cmbHeuristic.Name = "cmbHeuristic";
             addAStarHeuristicsToCmbBox(cmbHeuristic);
             cmbHeuristic.AutoSize = true;
             cmbHeuristic.Location = new Point(cmbAlgorithm.Location.X,
lblHeuristic.Location.Y);
             pnlScreenOptions.Controls.Add(cmbHeuristic);
        }

        private void addAStarHeuristicsToCmbBox(ComboBox cmb)
        {
             cmb.Items.Add("Manhattan");
             cmb.Items.Add("Euclidean");
             cmb.Items.Add("Chebyshev");

             cmb.Text = "Manhattan";       //Default
        }

         #endregion

         #region tabTilesMethods
```

```
        private void btnAddTileType_Click(object sender, EventArgs e)
        {
                /* ALGORITHM:
                * Create new panel, put in correct location, and add required controls
into it
                * Create new tileType
                * Increment panel counter by 1 (and also for unset panels)

                * Move pnlAddSet down to prepare for next new panel

                */
                //Note: Click on panel (not on its controls) to select a tileType to
place
                //     Click on pbx in panel to change tile colour

                string tbxOutput = "";

                Button btnAdd = (Button)sender;
                Panel pnlAddSet = (Panel)btnAdd.Parent;

                Panel pnlTileType = new Panel();
              pnlTileType.Size = new Size(tabCtrl.Width - 60, 70);
              pnlTileType.Location = new Point(30, pnlAddSet.Location.Y);

              pnlAddSet.Location = new Point(30, pnlAddSet.Location.Y + 70);  //Moves
add/set panel down
              addTileTypePanelCtrls(pnlTileType);

              pnlTileType.BackColor = Color.FromArgb(50, Color.Yellow);

                //Create new tileType
                string pnlNumber = (Global.lstDicTileTypeKeys.Count).ToString();
//Includes Empty and Wall - Direct comparison to list index
              pnlTileType.Name = "pnlTileType" + pnlNumber;
//Placeholder name
                Global.lstDicTileTypeKeys.Add(pnlTileType.Name);
                Global.dicTileTypeInfo.Add(pnlTileType.Name, new Tuple<Color,
int>(Color.BurlyWood, 0));  //Weight = 0 means tileType is not set

                //If one or more custom tileTypes exists, then btnSet is visible
                Button btnSet = (Button)pnlAddSet.Controls.Find("btnSetTileTypes",
false).FirstOrDefault();
              btnSet.Visible = isBtnSetVisible();

              tabTiles.Controls.Add(pnlTileType);

              tbxOutput = string.Format("Added new tileType");
              outputToTbxOutput(tbxOutput);

                Global.mapReadyToEdit = false;

                //Hide add button if max tile types reached
                if (Global.lstDicTileTypeKeys.Count() - 2 == Global.maxTileTypes)
                {
```

```
                btnAdd.Visible = false;

                tbxOutput = string.Format("Max tileTypes reached");
                outputToTbxOutput(tbxOutput);
              }
        }

         private void moveTileTypeSelector(PictureBox pbx)
        {
               Panel pnlClick = (Panel)pbx.Parent;
               Global.selectedTileType = pnlClick.Name;

               PictureBox pbxTileSelect =
(PictureBox)tabTiles.Controls.Find("pbxTileSelect", false).FirstOrDefault();
              pbxTileSelect.Location = new Point(pnlClick.Location.X - 22,
pnlClick.Location.Y + 20);

               string tbxOutput = "";
              tbxOutput = string.Format("Selected tileType changed to tileType '{0}'",
pnlClick.Name);
              outputToTbxOutput(tbxOutput);
        }

        private bool isBtnSetVisible()
        {
               bool isVisible = false;

               if (Global.lstDicTileTypeKeys.Count() > 2)
               {
                 isVisible = true;
               }

               return isVisible;
        }

        private void addTileTypePanelCtrls(Panel pnl)
        {
               //Draw new tile type
               PictureBox pbxNewTile = new PictureBox();
              pbxNewTile.Name = "pbxColour";
               pbxNewTile.Location = new Point(0, 0);
              pbxNewTile.Size = new Size(51, 51);     //FIXED - Includes grid line
              pbxNewTile.Tag = Color.BurlyWood;    //Default colour
              drawTileTypePbx(pbxNewTile, (Color)pbxNewTile.Tag);
              pbxNewTile.MouseClick += tileTypeColour_Click;           //Select tileType
/ Change tileType colour
              pnl.Controls.Add(pbxNewTile);

               //Draw labels and textboxes - Note: Setting backColor so changes to
backColor in panel doesn't affect those in child controls
               Label lblNewName = new Label();
              lblNewName.Text = "Name:      ";
              lblNewName.AutoSize = true;
              lblNewName.Location = new Point(70, 0);
```

```
            lblNewName.BackColor = Color.Transparent;
            pnl.Controls.Add(lblNewName);

              TextBox tbxNewName = new TextBox();
            tbxNewName.Name = "tbxName";
            tbxNewName.AutoSize = true;
            tbxNewName.Location = new Point(70 + lblNewName.Width, 0);
            tbxNewName.TextChanged += tbxTileType_TextChanged;       //To update
status of tileType to unset
            pnl.Controls.Add(tbxNewName);


              Label lblNewWeight = new Label();
            lblNewWeight.Text = "Weight: ";
            lblNewWeight.AutoSize = true;
            lblNewWeight.Location = new Point(70, 25);
            lblNewWeight.BackColor = Color.Transparent;
            pnl.Controls.Add(lblNewWeight);

              TextBox tbxNewWeight = new TextBox();
            tbxNewWeight.Name = "tbxWeight";
            tbxNewWeight.AutoSize = true;
            tbxNewWeight.Location = new Point(70 + lblNewWeight.Width, 25);
            tbxNewWeight.TextChanged += tbxTileType_TextChanged;     //To update
status of tileType to unset
            pnl.Controls.Add(tbxNewWeight);

              //Delete button
              Button btnDeleteTileType = new Button();
            btnDeleteTileType.Font = new Font("Microsoft Sans Serif", 13.0f,
FontStyle.Bold);
            btnDeleteTileType.Text = "x";
            btnDeleteTileType.Size = new Size(30, 30);
            btnDeleteTileType.Location = new Point(pnl.Width -
btnDeleteTileType.Width, 0);
            btnDeleteTileType.BackColor = Color.Transparent;
            pnl.Controls.Add(btnDeleteTileType);
            btnDeleteTileType.Click += btnDeleteTileType_Click;
        }

        private void tbxTileType_TextChanged(object sender, EventArgs e)
        {
              TextBox tbx = (TextBox)sender;
             Panel pnl = (Panel)tbx.Parent;

              if (pnl.BackColor != Color.FromArgb(50, Color.Yellow))
              {
                pnl.BackColor = Color.FromArgb(50, Color.Yellow);

                Global.mapReadyToEdit = false;

                string tbxOutput = "";
                tbxOutput = string.Format("Text change detected in tileType '{0}'.
Unsetting tileType", pnl.Name);
```

```
                outputToTbxOutput(tbxOutput);
            }
        }

        private void tileTypeColour_Click(object sender, MouseEventArgs e)
        {
            PictureBox pbx = (PictureBox)sender;

            //M1 Click - Select tileType
            if (e.Button == MouseButtons.Left)
            {
                Panel pnl = (Panel)pbx.Parent;
                if (pnl.BackColor == Color.FromArgb(70, Color.LawnGreen) || pnl.Name
== "Empty" || pnl.Name == "Wall")
                {
                    moveTileTypeSelector(pbx);
                }
            }

            //M2 Click - Change tileType colour
            else if (e.Button == MouseButtons.Right && pbx.Name != "pbxEmpty" &&
pbx.Name != "pbxWall")      //TEMPORARY: ONLY CUSTOM TILE TYPES CAN HAVE COLOUR
CHANGED
            {
                Color pbxOriginalColour = (Color)pbx.Tag;

                //Sets tileType colour based on chosen colour in clrPicker
                Color pbxNewColour;
                ColorDialog clrPicker = new ColorDialog();

                if (clrPicker.ShowDialog() == DialogResult.OK)
                {
                    pbxNewColour = clrPicker.Color;
                    pbx.Tag = pbxNewColour;      //Tag colour on pbx so we can refer to
it when updating
                    drawTileTypePbx(pbx, pbxNewColour);

                    //Updates panel status
                    if (pbxNewColour != pbxOriginalColour)
                    {
                     Panel pnl = (Panel)pbx.Parent;

                     string tbxOutput = "";
                     tbxOutput = string.Format("TileType '{0}' colour changed to
'{1}'", pnl.Name, pbxNewColour);
                        outputToTbxOutput(tbxOutput);

                     //If previously set and just updating tile colour, allow
insta-update
                     if (pnl.BackColor == Color.FromArgb(70, Color.LawnGreen))
                     {

updateTileType(Global.lstDicTileTypeKeys.IndexOf(pnl.Name), pnl.Name);
```

```
                        tbxOutput = string.Format("Updated colour of all tiles on
map with tileType '{0}'", pnl.Name);
                        outputToTbxOutput(tbxOutput);
                    }
                    }
                }
            }

        }

        private void btnSetTileTypes_Click(object sender, EventArgs e)
        {
            /* ALGORITHM:
            * For each custom panel in list
            *      If pass (Validation Check)
            *      Recreate dictionaries
            *      Update list key name
            *      Update panel name
              *          Change panel backcolour to green
            *      Recolour tiles of updated tileType
            *      Else
            *      Change panel backcolour to red
            *
            * If all panels green
            *      Map ready to edit
            */

            //RED = INVALID, YELLOW = PENDING, GREEN = VALID, SET  For back colours
            //(STILL NEED WEIGHTS = 0 THOUGH)

            string pnlKey = "";
            int pnlIndex = 2;    //Starts loop from start of custom tile types
(Note: No need to check if custom tileTypes exist as btnSet won't be visible)
            bool loopFinish = false;

            string tbxOutput = "";
          tbxOutput = string.Format("Setting unset tileTypes...");
          outputToTbxOutput(tbxOutput);

            //Need this as upper bound of loop will be dynamic
            while (loopFinish == false)
            {
              pnlKey = Global.lstDicTileTypeKeys[pnlIndex];
              updateTileType(pnlIndex, pnlKey);
              pnlIndex++;

              //If next index is beyond list bounds
              if (pnlIndex == Global.lstDicTileTypeKeys.Count)
              {
                  loopFinish = true;
              }
            }

          tbxOutput = string.Format("...valid unset tileTypes set");
```

```
            outputToTbxOutput(tbxOutput);

            checkMapReadyToEdit();
        }

        private void updateTileType(int pnlIndex, string pnlKey)
        {
                Panel pnl = (Panel)tabTiles.Controls.Find(pnlKey,
false).FirstOrDefault();
                string tbxOutput = "";

                //If not set and inputs are valid
                if (isTileTypeInputValid(pnlKey) == true)
                {
                tbxOutput = string.Format("Updating tileType '{0}'...", pnlKey);
                outputToTbxOutput(tbxOutput);

                TextBox tbxName = (TextBox)pnl.Controls.Find("tbxName",
false).FirstOrDefault();
                TextBox tbxWeight = (TextBox)pnl.Controls.Find("tbxWeight",
false).FirstOrDefault();
                     PictureBox pbxColour = (PictureBox)pnl.Controls.Find("pbxColour",
false).FirstOrDefault();

                Color prevColour = Global.dicTileTypeInfo[pnlKey].Item1;
                Color newColour = (Color)pbxColour.Tag;

                HashSet<string> hsetTiles = new HashSet<string>();

                //If tile to be updated is selected tile
                if (Global.selectedTileType == pnlKey)
                {
                        Global.selectedTileType = tbxName.Text;
                }

                //Update list of keys
                Global.lstDicTileTypeKeys[pnlIndex] = tbxName.Text;

                //If there is previous entry in dicTileTypeTiles - Move HashSet out,
delete dictionary entry
                if (Global.dicTileTypeTiles.ContainsKey(pnlKey))
                {
                        hsetTiles = Global.dicTileTypeTiles[pnlKey];
                        Global.dicTileTypeTiles.Remove(pnlKey);
                        //Update tileType values of existing tiles in map
                          foreach (string coords in hsetTiles)
                        {
                        Tile tileToUpdate = coordsToTile(coords);
                           tileToUpdate.setTileType(tbxName.Text);
                        }
                }

               //Create new dicTileTypeTiles entry
                Global.dicTileTypeTiles.Add(tbxName.Text, hsetTiles);
```

```
                //Recreate dicTileTypeInfo entry
                Global.dicTileTypeInfo.Remove(pnlKey);
                Global.dicTileTypeInfo.Add(tbxName.Text, new Tuple<Color,
int>((Color)pbxColour.Tag, int.Parse(tbxWeight.Text)));

                tbxOutput = string.Format("TileType '{0}' name changed to '{1}'",
pnlKey, tbxName.Text);
                outputToTbxOutput(tbxOutput);
                   tbxOutput = string.Format("TileType '{0}' weight set to {1}",
tbxName.Text, tbxWeight.Text);
                outputToTbxOutput(tbxOutput);

                if (prevColour != newColour)
                {
                       //Update pbx colour
                   drawTileTypePbx(pbxColour, (Color)pbxColour.Tag);

                       tbxOutput = string.Format("TileType '{0}' colour set to '{1}'",
tbxName.Text, pbxColour.Tag);
                      outputToTbxOutput(tbxOutput);

                       //Redraw existing tiles on grid
                       if (hsetTiles.Count() > 0)
                       {
                           recolourTiles(pnl.Name);
                       }
                }

                //Update panel
                pnl.Name = tbxName.Text;
                pnl.BackColor = Color.FromArgb(70, Color.LawnGreen);

                tbxOutput = string.Format("...tileType updated");
                outputToTbxOutput(tbxOutput);
                }
                else
                {
                pnl.BackColor = Color.FromArgb(50, Color.Red);

                Panel pnlWall = (Panel)tabTiles.Controls.Find("Wall",
false).FirstOrDefault();
                PictureBox pbxWall = (PictureBox)pnlWall.Controls.Find("pbxWall",
false).FirstOrDefault();
                    moveTileTypeSelector(pbxWall);
                }
        }

        private bool isTileTypeInputValid(string pnlKey)
        {
                string tbxOutput = "";
             tbxOutput = string.Format("Validating tileType '{0}'...", pnlKey);
             outputToTbxOutput(tbxOutput);
```

```
            Panel pnl = (Panel)tabTiles.Controls.Find(pnlKey, false).FirstOrDefault();
              TextBox tbxName = (TextBox)pnl.Controls.Find("tbxName",
false).FirstOrDefault();
              TextBox tbxWeight = (TextBox)pnl.Controls.Find("tbxWeight",
false).FirstOrDefault();

              //Check name
              bool nameValid = checkTileTypeName(tbxName.Text, pnl.Name);

              //Check weight
              bool weightValid = checkTileTypeWeight(tbxWeight.Text, pnl.Name);

              //Check if all inputs valid
              bool isValid = false;
              if (nameValid && weightValid)
              {
                isValid = true;
              }

              //Output to log

              if (isValid)
              {
                tbxOutput = string.Format("...tileType '{0}' is valid", pnl.Name);
              }
              else
              {
                tbxOutput = string.Format("...tileType '{0}' is not valid", pnl.Name);
              }
            outputToTbxOutput(tbxOutput);

              return isValid;
        }

        private bool checkTileTypeName(string tileTypeName, string pnlName)
        {
              bool nameValid = false;
              string tbxOutput = "";

              if (tileTypeName.ToUpper() != "EMPTY" && tileTypeName.ToUpper() !=
"WALL")       //Disallow name to be default names
              {
                //If updating weight only
                if (tileTypeName == pnlName)
                {
                    nameValid = true;
                }
                //If new name and tbx is blank
                else if (tileTypeName.Trim() != "")
                {
                    //If new name not taken
                    if (Global.lstDicTileTypeKeys.Contains(tileTypeName) == false)
                    {
                     nameValid = true;
```

```
                    }
                    else
                    {
                     tbxOutput = string.Format("tileType '{0}' has an already taken
name", pnlName);
                     outputToTbxOutput(tbxOutput);
                    }
                }
                else
                {
                    tbxOutput = string.Format("tileType '{0}' has blank name",
pnlName);
                    outputToTbxOutput(tbxOutput);
                }
            }
              else
              {
                tbxOutput = string.Format("tileType '{0}' has reserved name",
pnlName);
                outputToTbxOutput(tbxOutput);
              }

              return nameValid;
        }

        private bool checkTileTypeWeight(string tileTypeWeight, string pnlName)
        {
              bool weightValid = false;
              string tbxOutput = "";

              //Check if numeric
              int parseResult = 0;
              if (int.TryParse(tileTypeWeight, out parseResult))
              {
                //Check if within bounds
                if (parseResult > 0)
                {
                    weightValid = true;
                }
                else
                {
                    tbxOutput = string.Format("tileType '{0}' has negative weight",
pnlName);
                    outputToTbxOutput(tbxOutput);
                }
              }
              else
              {
                tbxOutput = string.Format("tileType '{0}' has non-integer weight",
pnlName);
                outputToTbxOutput(tbxOutput);
              }

              return weightValid;
```

```
        }

        private void recolourTiles(string tileTypeKey)
        {
                HashSet<string> tileCoords = Global.dicTileTypeTiles[tileTypeKey];
              updateTilesOnMap(tileCoords);
        }

        private void btnDeleteTileType_Click(object sender, EventArgs e)
        {
                /* ALGORITHM:
                *  Notify user that tiles of this type on grid will be deleted
                *If 'OK',
                *   Delete tiles from map (if any) - Set to default
                *   Obtain indexToDelete by [(btnDelete.Y - 1stpnl.TopLeft.Y) /
pnlHeight]    {1st pnlTopLeft = 190}
                *   Move pnlAddSet up
                *   Delete panel (parent of btnDelete) from tabTiles
                *   Decrement panel counter by 1
                *   Use indexToDelete to delete key from dictionary
                *   For all index higher than (index to delete) in list
                *      Move panel upwards in tabTiles
                *      Rename unset panels that have moved and recreate in dictionary

                *      Move panel index in list by 1 towards 0
                *   Delete last element of list of keys (lst.Count - 1)

                *   Determine if btnSet will be visible or not
                */
                Button btnDelete = (Button)sender;
                Panel pnlToDelete = (Panel)btnDelete.Parent;
                Panel pnlAddSet = (Panel)tabTiles.Controls.Find("pnlAddSet",
false).FirstOrDefault();

                string tbxOutput = "";
              tbxOutput = string.Format("Deleting tileType '{0}'...", pnlToDelete.Name);
              outputToTbxOutput(tbxOutput);

                //Make btnAdd in pnlAddSet to be visible again
                Button btnAdd = (Button)pnlAddSet.Controls.Find("btnAddTileType",
false).FirstOrDefault();
              btnAdd.Visible = true;

                string keyToDelete = pnlToDelete.Name;
                int indexToDelete = Global.lstDicTileTypeKeys.IndexOf(pnlToDelete.Name);

                //Moves add/set panel up
              pnlAddSet.Location = new Point(30, pnlAddSet.Location.Y - 70);

                //If tileTypeSelector is on tileTypes to move, move tileTypeSelector to
'Empty' tileType
                int indexSelectedTile =
Global.lstDicTileTypeKeys.IndexOf(Global.selectedTileType);
                if (indexSelectedTile >= indexToDelete)
```

```
            {
                Panel pnlWall = (Panel)tabTiles.Controls.Find("Wall",
false).FirstOrDefault();
                PictureBox pbxWall = (PictureBox)pnlWall.Controls.Find("pbxWall",
false).FirstOrDefault();
                moveTileTypeSelector(pbxWall);
            }

            //Delete panel
            tabTiles.Controls.Remove(pnlToDelete);

            //If tileType previously set
            if (Global.dicTileTypeTiles.ContainsKey(keyToDelete))
            {
                //Set tiles of to-be-deleted tileType to "Empty" tileType
                foreach (string tileCoords in Global.dicTileTypeTiles[keyToDelete])
                {
                    Tile tileToDelete = coordsToTile(tileCoords);
                    tileToDelete.setTileType("Empty");
                }

                recolourTiles(keyToDelete);
                Global.dicTileTypeTiles.Remove(keyToDelete);
            }

            //Delete tileTypeInfo from dictionary
            Global.dicTileTypeInfo.Remove(keyToDelete);

            //Reorganise list, dict, and panels in tabTiles
            reorganiseTileTypes(indexToDelete);

            //If one or more custom tileTypes exists, then btnSet is visible
            Button btnSet = (Button)pnlAddSet.Controls.Find("btnSetTileTypes",
false).FirstOrDefault();
            btnSet.Visible = isBtnSetVisible();

            tbxOutput = string.Format("...tileType '{0}' deleted", keyToDelete);
            outputToTbxOutput(tbxOutput);

            checkMapReadyToEdit();
        }

        private void reorganiseTileTypes(int indexToDelete)
        {
            for (int i = indexToDelete; i < Global.lstDicTileTypeKeys.Count() - 1;
i++)
            {
                Panel pnlToMove =
(Panel)tabTiles.Controls.Find(Global.lstDicTileTypeKeys[i + 1],
false).FirstOrDefault();

                //Recreating key for dictionary - If unset
                if (Global.dicTileTypeInfo[pnlToMove.Name].Item2 == 0)
                {
```

```
                    //Deletes old name
                    Global.dicTileTypeInfo.Remove(pnlToMove.Name);

                    //Generates new name based on new index in list and recreates
dictionary key
                    string pnlNumber = i.ToString();
                    pnlToMove.Name = "pnlTileType" + pnlNumber;
                    Global.dicTileTypeInfo.Add(pnlToMove.Name, new Tuple<Color,
int>(Color.BurlyWood, 0));
                }

                //Shifting index in list
                Global.lstDicTileTypeKeys[i] = pnlToMove.Name;

                //Moves panel up 1 slot
                pnlToMove.Location = new Point(30, pnlToMove.Location.Y - 70);
            }

            //Remove last element in list
            Global.lstDicTileTypeKeys.RemoveAt(Global.lstDicTileTypeKeys.Count() -
1);

            string tbxOutput = "";
          tbxOutput = string.Format("Tiletypes reorganised");
          outputToTbxOutput(tbxOutput);
      }

      private void checkMapReadyToEdit()
      {
            if (!Global.donePathfinding)
            {
              //Assumes ready to edit until non-green panel found
              Global.mapReadyToEdit = true;
              string tbxOutput = "";

              //Loops through all tileTypes (excluding defaults)
              for (int i = 2; i < Global.lstDicTileTypeKeys.Count(); i++)
              {
                  string pnlName = Global.lstDicTileTypeKeys[i];
                  Panel pnl = (Panel)tabTiles.Controls.Find(pnlName,
false).FirstOrDefault();

                  if (pnl.BackColor != Color.FromArgb(70, Color.LawnGreen))
                  {
                   Global.mapReadyToEdit = false;
                   i = Global.lstDicTileTypeKeys.Count() - 1;
                  }
              }

                if (Global.mapReadyToEdit)
              {
                  tbxOutput = string.Format("Map ready to edit");
              }
              else
```

```
                {
                    tbxOutput = string.Format("Map not ready to edit");
                }
                outputToTbxOutput(tbxOutput);
            }
        }
         #endregion

#endregion
```

BestFitScreen

The code below is an implementation of the best-fit picturebox algorithm described in the design section:

```
#region BestFitScreen
       private void BestFitPbx()
       {
              //Note: This method fits screens, regardless of tileLength
              Tuple<int, string> tileLengthStatus = trialFitScreen();
              Global.tileLength = tileLengthStatus.Item1;
            placeScreens();
       }

       private Tuple<int, string> trialFitScreen()
       {
              string tbxOutput = "";
            tbxOutput = string.Format("Trial fitting screens by attempting to create
enough slots...");
            outputToTbxOutput(tbxOutput);

              //Note: 'lastAction' is used to determine whether width/height/both
caused tileLength to go below minimum - For 1 screen only

              //To stop 'fitting' if tileLength goes below minimum
              bool belowMinTilelength = false;

              Tuple<int, string> tileLengthStatus = getInitTileLength();
              int testTileLength = tileLengthStatus.Item1;
              string lastAction = tileLengthStatus.Item2;

              //Initial tileLength checker
              if (testTileLength < Global.minTileLength)
              {
                belowMinTilelength = true;

                tbxOutput = string.Format("Trial tileLength has gone less than minimum
tileLength");
                outputToTbxOutput(tbxOutput);
              }
              else
              {
                //Initial total number of slots
                int slotLength = (Global.noOfTilesX * testTileLength) +
Global.borderLength + 1;
                int layerLength = (Global.noOfTilesY * testTileLength) +
Global.borderLength + 1;
```

```
                int slotsPerLayer = (int)Math.Floor((decimal)pnlGrid.Width /
slotLength);
                int noOfLayers = (int)Math.Floor((decimal)pnlGrid.Height /
layerLength);

                int totNoOfSlots = slotsPerLayer * noOfLayers;

                //Gradually adds slots to pnlGrid until enough to fit all screens in
                while (totNoOfSlots < Global.lstScreens.Count() && belowMinTilelength
== false)
                {
                    int slotTileLength = (int)Math.Floor((1 /
(decimal)Global.noOfTilesX) * (((decimal)pnlGrid.Width / (slotsPerLayer + 1)) -
Global.borderLength - 1));
                    int layerTileLength = (int)Math.Floor((1 /
(decimal)Global.noOfTilesY) * (((decimal)pnlGrid.Height / (noOfLayers + 1)) -
Global.borderLength - 1));

                    //Finds out whether new slot or new layer comes first as
tileLength decreases and sets tileLength this value
                    if (slotTileLength > layerTileLength)
                    {
                     testTileLength = slotTileLength;
                     totNoOfSlots += noOfLayers;
                     slotsPerLayer++;
                    }
                    else if (slotTileLength < layerTileLength)
                    {
                     testTileLength = layerTileLength;
                     totNoOfSlots += slotsPerLayer;
                     noOfLayers++;
                    }
                    else if (slotTileLength == layerTileLength)
                    {
                     testTileLength = slotTileLength;
                     slotsPerLayer++;
                     totNoOfSlots += noOfLayers;
                     noOfLayers++;
                     totNoOfSlots += slotsPerLayer;
                    }

                    //TileLength checker
                    if (testTileLength < Global.minTileLength)
                    {
                     belowMinTilelength = true;

                     tbxOutput = string.Format("Trial tileLength has gone less than
minimum tileLength");
                        outputToTbxOutput(tbxOutput);
                    }
                }
            }

            if (belowMinTilelength)
```

```
            {
              //Failed = -1, Pass = Some +ve val > minTileLength
              testTileLength = -1;

              tbxOutput = string.Format("...trial fitting failed");
              outputToTbxOutput(tbxOutput);
            }
            else
            {
              tbxOutput = string.Format("...trial fitting succeeded");
              outputToTbxOutput(tbxOutput);
            }

          tileLengthStatus = new Tuple<int, string>(testTileLength, lastAction);
            return tileLengthStatus;
      }

      private void placeScreens()
      {
            //Creates and places screens into slots
            int slotLength = (Global.noOfTilesX * Global.tileLength) +
Global.borderLength + 1;
            int layerLength = (Global.noOfTilesY * Global.tileLength) +
Global.borderLength + 1;
             int slotsPerLayer = (int)Math.Floor((decimal)pnlGrid.Width /
slotLength);
            int slotX = 0;
            int slotY = 0;

            //Clears existing screens
          pnlGrid.Controls.Clear();
            for (int i = 0; i < Global.lstScreens.Count(); i++)
            {
              Point screenLocation = new Point(slotX * slotLength, slotY *
layerLength);
              Size screenSize = new Size(slotLength, layerLength);

              Panel pnlScreen = createScreen("Screen" + i, screenLocation,
screenSize);
              pnlGrid.Controls.Add(pnlScreen);

              //Moves to next slot if not last screen
              if (i < Global.lstScreens.Count() - 1)
              {
                  slotX++;
                   if (slotX == slotsPerLayer)
                  {
                   slotY++;
                   slotX = 0;
                  }
              }
            }

            //Update maxNoOfTilesXY for current screen state
```

```
            //Note: Equations derived from noOfScreensX = pnlWidth / screenWidth
            int noOfLayers = (int)Math.Ceiling((decimal)Global.lstScreens.Count() /
slotsPerLayer);
            Global.maxNoOfTilesX = (int)Math.Floor(((decimal)1 /
Global.minTileLength) * (((decimal)pnlGrid.Width / slotsPerLayer) -
Global.borderLength - 1));
            Global.maxNoOfTilesY = (int)Math.Floor(((decimal)1 /
Global.minTileLength) * (((decimal)pnlGrid.Height / noOfLayers) - Global.borderLength
- 1));

            string tbxOutput = "";
          tbxOutput = string.Format("Screens placed into slots");
          outputToTbxOutput(tbxOutput);
      }
#endregion
```

DrawGrid

The code below describes how graphics are to be drawn on the screen and the overlay. The code features heavy use of lockbits, which is used to directly edit bitmaps by locking it into system memory and then using pointers, via 'unsafe code', to access its bytes. When editing the bytes, I have found that endianness matters. As the computers used to work on this project are in little-endian format, the colours had to be accessed in the order BGRA rather than ARGB. I have also coded for computers using big-endian for completion's sake.

```
#region DrawGrid
       private void DrawFullGrid()
       {
              string tbxOutput = "";
            tbxOutput = string.Format("Drawing full grid to screen...");
            outputToTbxOutput(tbxOutput);

              Bitmap bmp = new Bitmap((Global.noOfTilesX * Global.tileLength) + 1,
(Global.noOfTilesY * Global.tileLength) + 1);
            Graphics gfx = Graphics.FromImage(bmp);

              //For each tile to draw
              for (int row = 0; row < Global.noOfTilesY; row++)
              {
                for (int column = 0; column < Global.noOfTilesX; column++)
                {
                    Rectangle rect = new Rectangle((column * Global.tileLength), (row
* Global.tileLength), Global.tileLength, Global.tileLength);

                    //Obtain colour
                    Tile tempTile = Global.lstMapRow[row][column];
                    Color tileColour = tempTile.getColour();

                    //Colour rectangle
                    gfx.FillRectangle(new SolidBrush(tileColour), rect);

                    //Grid Lines
                    gfx.DrawRectangle(Pens.Black, rect);
                }
              }

              //Output bmp to all screens
              foreach (Panel pnlScreen in pnlGrid.Controls)
              {
                PictureBox pbxGrid = (PictureBox)pnlScreen.Controls[0];
                pbxGrid.Image = bmp;

                //Creates overlay - Map resize
                initPbxGridOverlay(pbxGrid);
              }

            tbxOutput = string.Format("...full grid drawn to screen");
            outputToTbxOutput(tbxOutput);
```

```
}

private void initPbxGridOverlay(PictureBox pbxGrid)
{
       Panel pnl = (Panel)pbxGrid.Parent;

       string tbxOutput = "";
     tbxOutput = string.Format("Initialising {0}'s overlay...", pnl.Name);
     outputToTbxOutput(tbxOutput);

       PictureBox pbxOverlay = new PictureBox();
     pbxOverlay.Name = "pbxOverlay";
     pbxOverlay.BackColor = Color.Transparent;
     pbxOverlay.Location = new Point(0, 0);
     pbxOverlay.Size = pbxGrid.Size;

     pbxOverlayAddHandlers(pbxOverlay);

       //If overlay exists, remove before adding new one
       if (pbxGrid.Controls.Find("pbxOverlay", false).FirstOrDefault() != null)
       {
         //Note: pbxGrid will always only have one child control
         pbxGrid.Controls.RemoveAt(0);

         tbxOutput = string.Format("Removed existing overlay");
         outputToTbxOutput(tbxOutput);
       }
     pbxGrid.Controls.Add(pbxOverlay);
     tbxOutput = string.Format("Added overlay to screen");
     outputToTbxOutput(tbxOutput);

       //Initialises bitmap that pbxOverlay will hold
       Bitmap bmpOverlay = new Bitmap(pbxGrid.Width, pbxGrid.Height);
     tbxOutput = string.Format("Created empty bitmap for overlay");
     outputToTbxOutput(tbxOutput);

       //Add 'X's to overlay
     fillPbxOverlay(bmpOverlay);

       //Initial values for tile highlight - Off-overlay
       int offsetX = pnlGrid.Location.X + pbxGrid.Location.X;
       int offsetY = pnlGrid.Location.Y + pbxGrid.Location.Y;
       Global.curTileHighlightX = MousePosition.X - offsetX;
       Global.curTileHighlightY = MousePosition.Y - offsetY;

     pbxOverlay.Image = bmpOverlay;
     tbxOutput = string.Format("...overlay initialised");
     outputToTbxOutput(tbxOutput);
}

private void pbxOverlayAddHandlers(PictureBox pbxOverlay)
{
   pbxOverlay.MouseEnter += overlay_MouseEnter;
     pbxOverlay.MouseLeave += overlay_MouseLeave;
```

```
            pbxOverlay.MouseMove += overlay_MouseMove;
            pbxOverlay.MouseClick += overlay_MouseClick;      //For clicking but no
mouse movement
        }

        private void fillPbxOverlay(Bitmap bmpOverlay)
        {
               foreach (string coords in Global.lstPointsToPathfind)
               {
                 drawXOnOverlay(bmpOverlay, coords);
               }

               string tbxOutput = "";
            tbxOutput = string.Format("Drawn path points on new overlay");
            outputToTbxOutput(tbxOutput);
        }

        private void drawXOnOverlay(Bitmap bmp, string coords)
        {
               //Lock bmp bits to sys mem
               Rectangle rect = new Rectangle(0, 0, bmp.Width, bmp.Height);
               BitmapData bmpData = bmp.LockBits(rect, ImageLockMode.ReadWrite,
PixelFormat.Format32bppArgb);

               //32-bits - Each mem location holds 1 byte
               int Bpp = 4;

              drawXOnTile(ref bmpData, Bpp, coords);

               //NOTE: UPDATING METHOD (IGNORING GRID LINES) MUST BE UPDATED IF OPTION
TO DISCLUDE GRID LINES ADDED

            bmp.UnlockBits(bmpData);
        }

        private void drawXOnTile(ref BitmapData bmpData, int Bpp, string coords)
        {
               Point tileCoords = coordsStringToPoint(coords);

               unsafe
               {
                 //Get address of first line
                 byte* ptrStart = (byte*)bmpData.Scan0;

                 //Location of tile in memory - Note: Think very long 1D array of bytes
                 byte* tileRow = ptrStart + (tileCoords.Y * Global.tileLength *
bmpData.Stride);
                 int tileTopLeft = tileCoords.X * Global.tileLength * Bpp;
//Relative to tileRow

                 // Drawing '\' part of cross
                 for (int i = 1; i < Global.tileLength; i++)
                 {
                     int offsetY = i * bmpData.Stride;
```

```
                int offsetX = i * Bpp;
                int totOffset = offsetX + offsetY;

                //Big endian v.s. Little endian byte order
                if (BitConverter.IsLittleEndian)
                {
                 tileRow[(tileTopLeft) + totOffset] = Color.Black.B;
                   tileRow[(tileTopLeft + 1) + totOffset] = Color.Black.G;
                 tileRow[(tileTopLeft + 2) + totOffset] = Color.Black.R;
                 tileRow[(tileTopLeft + 3) + totOffset] = Color.Black.A;
                }
                else
                {
                 tileRow[(tileTopLeft) + totOffset] = Color.Black.A;
                 tileRow[(tileTopLeft + 1) + totOffset] = Color.Black.R;
                 tileRow[(tileTopLeft + 2) + totOffset] = Color.Black.G;
                 tileRow[(tileTopLeft + 3) + totOffset] = Color.Black.B;
                }
            }

            // Drawing '/' part of cross
            for (int i = 1; i < Global.tileLength; i++)
            {
                int offsetY = i * bmpData.Stride;
                int offsetX = (Global.tileLength - i) * Bpp;
                int totOffset = offsetX + offsetY;

                //Big endian v.s. Little endian byte order
                if (BitConverter.IsLittleEndian)
                {
                 tileRow[(tileTopLeft) + totOffset] = Color.Black.B;
                 tileRow[(tileTopLeft + 1) + totOffset] = Color.Black.G;
                 tileRow[(tileTopLeft + 2) + totOffset] = Color.Black.R;
                 tileRow[(tileTopLeft + 3) + totOffset] = Color.Black.A;
                }
                else
                {
                 tileRow[(tileTopLeft) + totOffset] = Color.Black.A;
                 tileRow[(tileTopLeft + 1) + totOffset] = Color.Black.R;
                 tileRow[(tileTopLeft + 2) + totOffset] = Color.Black.G;
                 tileRow[(tileTopLeft + 3) + totOffset] = Color.Black.B;
                }
            }
        }
}

private void updateTileOnMap(string coords)
{
        //Take first pbx
        Panel pnlScreen = (Panel)pnlGrid.Controls[0];
        PictureBox pbxGrid = (PictureBox)pnlScreen.Controls[0];
        Bitmap bmp = (Bitmap)pbxGrid.Image;

        //Lock bmp bits to sys mem
```

```
            Rectangle rect = new Rectangle(0, 0, bmp.Width, bmp.Height);
            BitmapData bmpData = bmp.LockBits(rect, ImageLockMode.ReadWrite,
PixelFormat.Format24bppRgb);

            //24-bits - Each mem location holds 1 byte
            int Bpp = 3;

          fillTileOnMap(ref bmpData, Bpp, coords);

          //NOTE: UPDATING METHOD (IGNORING GRID LINES) MUST BE UPDATED IF OPTION TO
DISCLUDE GRID LINES ADDED

          bmp.UnlockBits(bmpData);
          pnlGrid.Refresh();
      }

      private void fillTileOnMap(ref BitmapData bmpData, int Bpp, string coords)
      {
            Point tileCoords = coordsStringToPoint(coords);

            //Get tile colour - Colour depends on status
            Tile tileToUpdate = coordsToTile(coords);
            Color tileColour = tileToUpdate.getColour();

            unsafe
            {
              //Get address of first line
              byte* ptrStart = (byte*)bmpData.Scan0;

              //Location of tile in memory - Note: Think very long 1D array of bytes
               byte* tileRow = ptrStart + (tileCoords.Y * Global.tileLength *
bmpData.Stride);
              int tileTopLeft = tileCoords.X * Global.tileLength * Bpp;
//Relative to tileRow

              //Row
              for (int i = 0; i < Global.tileLength; i++)
              {
                  //Column
                  for (int j = 0; j < Global.tileLength; j++)
                  {
                   int offsetY = i * bmpData.Stride;
                   int offsetX = j * Bpp;
                   int totOffset = offsetX + offsetY;

                   //Tile Colour - Only draw inside grid lines
                   if (i != 0 && i != Global.tileLength && j != 0 && j !=
Global.tileLength)
                   {
                         //Big endian v.s. Little endian byte order
                         if (BitConverter.IsLittleEndian)
                         {
                              tileRow[(tileTopLeft) + totOffset] = tileColour.B;
```

```
                                    tileRow[(tileTopLeft + 1) + totOffset] =
tileColour.G;
                                tileRow[(tileTopLeft + 2) + totOffset] = tileColour.R;
                            }
                            else
                            {
                                tileRow[(tileTopLeft) + totOffset] = tileColour.R;
                                tileRow[(tileTopLeft + 1) + totOffset] = tileColour.G;
                                tileRow[(tileTopLeft + 2) + totOffset] = tileColour.R;
                            }
                        }
                       }
                  }
                }
        }

        private void updateTilesOnMap(HashSet<string> hsetTilesToUpdate)
        {
                //Take first pbx
                Panel pnlScreen = (Panel)pnlGrid.Controls[0];
                PictureBox pbxGrid = (PictureBox)pnlScreen.Controls[0];
                Bitmap bmp = (Bitmap)pbxGrid.Image;

                //Lock bmp bits to sys mem
                Rectangle rect = new Rectangle(0, 0, bmp.Width, bmp.Height);
                BitmapData bmpData = bmp.LockBits(rect, ImageLockMode.ReadWrite,
PixelFormat.Format24bppRgb);

                //24-bits - Each mem location holds 1 byte
                int Bpp = 3;

                //Looping through all tiles to update
                foreach (string coords in hsetTilesToUpdate)
                {
                  fillTileOnMap(ref bmpData, Bpp, coords);
                }

                //NOTE: UPDATING METHOD (IGNORING GRID LINES) MUST BE UPDATED IF OPTION
TO DISCLUDE GRID LINES ADDED

               bmp.UnlockBits(bmpData);
               pnlGrid.Refresh();
        }

        //Draws pathfinding status on overlay
        private void setOverlayTileStatus(PictureBox pbxOverlay, string coords, Color
tileColour)
        {
           Bitmap bmp = (Bitmap)pbxOverlay.Image;

                //Lock bmp bits to sys mem
                Rectangle rect = new Rectangle(0, 0, bmp.Width, bmp.Height);
                BitmapData bmpData = bmp.LockBits(rect, ImageLockMode.ReadWrite,
PixelFormat.Format32bppArgb);
```

```
                //32-bits - Each mem location holds 1 byte
                int Bpp = 4;

                Point tileCoords = coordsStringToPoint(coords);

                unsafe
                {
                  //Get address of first line
                  byte* ptrStart = (byte*)bmpData.Scan0;

                  //Location of tile in memory - Note: Think very long 1D array of bytes
                  byte* tileRow = ptrStart + (tileCoords.Y * Global.tileLength *
bmpData.Stride);
                  int tileTopLeft = tileCoords.X * Global.tileLength * Bpp;
//Relative to tileRow

                  //Row
                  for (int i = 0; i < Global.tileLength; i++)
                  {
                      //Column
                      for (int j = 0; j < Global.tileLength; j++)
                      {
                       int offsetY = i * bmpData.Stride;
                       int offsetX = j * Bpp;
                       int totOffset = offsetX + offsetY;

                       //Tile Colour - Only draw inside grid lines
                       if (i != 0 && i != Global.tileLength && j != 0 && j !=
Global.tileLength)
                       {
                              //Big endian v.s. Little endian byte order
                              if (BitConverter.IsLittleEndian)
                              {
                                   tileRow[(tileTopLeft) + totOffset] = tileColour.B;
                                   tileRow[(tileTopLeft + 1) + totOffset] = tileColour.G;
                                     tileRow[(tileTopLeft + 2) + totOffset] =
tileColour.R;
                                   tileRow[(tileTopLeft + 3) + totOffset] = tileColour.A;
                              }
                              else
                              {
                                  tileRow[(tileTopLeft) + totOffset] = tileColour.A;
                                   tileRow[(tileTopLeft + 1) + totOffset] = tileColour.R;
                                   tileRow[(tileTopLeft + 2) + totOffset] = tileColour.G;
                                    tileRow[(tileTopLeft + 3) + totOffset] =
tileColour.B;
                              }
                       }
                      }
                  }

                  //NOTE: UPDATING METHOD (IGNORING GRID LINES) MUST BE UPDATED IF
OPTION TO DISCLUDE GRID LINES ADDED
```

```
                bmp.UnlockBits(bmpData);
                pbxOverlay.Image = bmp;

                if (Global.fastSearch == false)
                {
                    pbxOverlay.Update();        //Goes really fast if this is disabled
                    System.Threading.Thread.Sleep(500);       //Slo-mo
                }
            }
        }

        private Color getOverlayTileStatus(PictureBox pbxOverlay, string coords)
        {
            Bitmap bmp = (Bitmap)pbxOverlay.Image;

            //Lock bmp bits to sys mem
            Rectangle rect = new Rectangle(0, 0, bmp.Width, bmp.Height);
            BitmapData bmpData = bmp.LockBits(rect, ImageLockMode.ReadWrite,
PixelFormat.Format32bppArgb);

            //32-bits - Each mem location holds 1 byte
            int Bpp = 4;

            Point tileCoords = coordsStringToPoint(coords);

            unsafe
            {
                //Get address of first line
                byte* ptrStart = (byte*)bmpData.Scan0;

                //Location of tile in memory - Note: Think very long 1D array of bytes
                byte* tileRow = ptrStart + (tileCoords.Y * Global.tileLength *
bmpData.Stride);
                int tileTopLeft = tileCoords.X * Global.tileLength * Bpp;
//Relative to tileRow

                //Take tile colour @ (1, 2) from grid line top left - Will always
contian colour incase of X
                int offsetY = 2 * bmpData.Stride;
                int offsetX = Bpp;
                int totOffset = offsetX + offsetY;
                byte A, R, G, B;

                //Big endian v.s. Little endian byte order
                if (BitConverter.IsLittleEndian)
                {
                    B = tileRow[(tileTopLeft) + totOffset];
                       G = tileRow[(tileTopLeft + 1) + totOffset];
                    R = tileRow[(tileTopLeft + 2) + totOffset];
                    A = tileRow[(tileTopLeft + 3) + totOffset];
                }
                else
                {
```

```
                    A = tileRow[(tileTopLeft) + totOffset];
                R = tileRow[(tileTopLeft + 1) + totOffset];
                G = tileRow[(tileTopLeft + 2) + totOffset];
                B = tileRow[(tileTopLeft + 3) + totOffset];
            }

            bmp.UnlockBits(bmpData);

            Color tileColour = Color.FromArgb(A, R, G, B);
            return tileColour;
        }
    }

#endregion
```

MouseFunctions

The code below contains the methods that will be executed when certain mouse buttons are clicked at certain areas of the program:

```
#region MouseFunctions

        private void replaceTileHighLight(PictureBox pbxOverlay)
        {
                Bitmap bmpOverlay = (Bitmap)pbxOverlay.Image;
                Graphics gfx = Graphics.FromImage(bmpOverlay);
             gfx.CompositingMode = System.Drawing.Drawing2D.CompositingMode.SourceCopy;
        //Replaces pixels rather than blend

                //Get highlighted old tile map coordinates
                int tileX = Global.curTileHighlightX / Global.tileLength;
                int tileY = Global.curTileHighlightY / Global.tileLength;
                string tileCoords = tileX + "," + tileY;

                //Delete previous highlight
               gfx.DrawRectangle(Pens.Transparent, Global.curTileHighlightX,
Global.curTileHighlightY, Global.tileLength, Global.tileLength);

                Global.curTileHighlightX = Global.newTileHighlightX;
                Global.curTileHighlightY = Global.newTileHighlightY;

                //Only draw new tile highlight if within map bounds
                if (Global.curTileHighlightX <= pbxOverlay.Width &&
Global.curTileHighlightY <= pbxOverlay.Height)
                {
                  //Get highlighted new tile map coordinates
                  tileX = Global.curTileHighlightX / Global.tileLength;
                  tileY = Global.curTileHighlightY / Global.tileLength;
                  tileCoords = tileX + "," + tileY;

                  //Draw over with highlight
                  gfx.DrawRectangle(Pens.Red, Global.curTileHighlightX,
Global.curTileHighlightY, Global.tileLength, Global.tileLength);
                }

             pbxOverlay.Image = bmpOverlay;
        }

        private void doMouseButtonPressedEvents(MouseEventArgs e)
        {
                //tabScreen open
                if (tabCtrl.SelectedIndex == 0 && Global.allowEditPoints)
                {
                   //M1 pressed
                  if (e.Button == MouseButtons.Left)
                  {
                      addPathfindingPoint();
                  }
```

```
                //M2 pressed
                else if (e.Button == MouseButtons.Right)
                {
                     deletePathfindingPoint();
                }
            }
            //tabTiles open
            else if (tabCtrl.SelectedIndex == 1 && Global.mapReadyToEdit)
            {
                //M1 pressed
                if (e.Button == MouseButtons.Left)
                {
                    setHighlightedTileType(Global.selectedTileType);
                }
                //M2 pressed
                else if (e.Button == MouseButtons.Right)
                {
                    setHighlightedTileType("Empty");
                }
            }
        }

        private void addPathfindingPoint()
        {
            //PBXGRID = PBX THAT MOUSE IS ABOVE - BASE ON OVERLAY THAT MOUSE IS
ABOVE, AND GET PARENT
            Panel pnlScreen = (Panel)pnlGrid.Controls[0];
            PictureBox pbxGrid = (PictureBox)pnlScreen.Controls[0];
            PictureBox pbxOverlay = (PictureBox)pbxGrid.Controls[0];

            //Set location = Highlighted tile
            int tileX = Global.curTileHighlightX / Global.tileLength;
            int tileY = Global.curTileHighlightY / Global.tileLength;

            Tile tileSelected = Global.lstMapRow[tileY][tileX];

            //Disallows pathfinding to wall - Impassable
            if (tileSelected.getTileType() != "Wall")
            {
                string tileCoords = tileSelected.getCoords();

                //Limits number of points to pathfind between
                if (Global.lstPointsToPathfind.Count() < Global.maxPathPoints &&
!(Global.lstPointsToPathfind.Contains(tileCoords)))
                {
                    Global.lstPointsToPathfind.Add(tileCoords);

                    //Draw X on all overlays
                    foreach (Panel screen in pnlGrid.Controls)
                    {
                     PictureBox grid = (PictureBox)screen.Controls[0];
                     PictureBox overlay = (PictureBox)grid.Controls[0];
                     Bitmap bmpOverlay = (Bitmap)overlay.Image;
                         drawXOnOverlay(bmpOverlay, tileCoords);
```

```
                    }
                    pnlGrid.Refresh();

                    string tbxOutput = "";
                    string[] tempOutput = tileCoords.Split(',');
                    tbxOutput = string.Format("Path point drawn on coordinate ({0},
{1})", tileCoords[0], tileCoords[1]);
                    outputToTbxOutput(tbxOutput);
                }
            }
        }

        private void deletePathfindingPoint()
        {
            //PBXGRID = PBX THAT MOUSE IS ABOVE - BASE ON OVERLAY THAT MOUSE IS
ABOVE, AND GET PARENT
            Panel pnlScreen = (Panel)pnlGrid.Controls[0];
            PictureBox pbxGrid = (PictureBox)pnlScreen.Controls[0];
            PictureBox pbxOverlay = (PictureBox)pbxGrid.Controls[0];
            Bitmap bmpOverlay = (Bitmap)pbxOverlay.Image;

            //Set location = Highlighted tile
            int tileX = Global.curTileHighlightX / Global.tileLength;
            int tileY = Global.curTileHighlightY / Global.tileLength;

            Tile tileSelected = Global.lstMapRow[tileY][tileX];
            string tileCoords = tileSelected.getCoords();

            //Check if tile has path point
            if (Global.lstPointsToPathfind.Contains(tileCoords))
             {
              int deletedIndex = Global.lstPointsToPathfind.IndexOf(tileCoords);
              reorderStringList(ref Global.lstPointsToPathfind, deletedIndex);

              //Erases X on all overlays
              foreach (Panel screen in pnlGrid.Controls)
              {
                  PictureBox grid = (PictureBox)screen.Controls[0];
                  PictureBox overlay = (PictureBox)grid.Controls[0];
                  Bitmap bmp = (Bitmap)overlay.Image;

                  //Erases cross
                  Graphics gfx = Graphics.FromImage(bmp);
                  gfx.CompositingMode =
System.Drawing.Drawing2D.CompositingMode.SourceCopy;   //Replaces pixels rather than
blend
                  gfx.FillRectangle(Brushes.Transparent, tileX * Global.tileLength +
1, tileY * Global.tileLength + 1, Global.tileLength - 1, Global.tileLength - 1);
              }
              pnlGrid.Refresh();

              string tbxOutput = "";
              string[] tempOutput = tileCoords.Split(',');
```

```
                tbxOutput = string.Format("Path point at coordinate ({0}, {1})
deleted", tempOutput[0], tempOutput[1]);
                outputToTbxOutput(tbxOutput);
            }
        }

        private void setHighlightedTileType(string targetTileType)
        {
            //Tile to set = Highlighted tile
            int tileX = Global.curTileHighlightX / Global.tileLength;
            int tileY = Global.curTileHighlightY / Global.tileLength;

            Tile tileToUpdate = Global.lstMapRow[tileY][tileX];
            setTileType(tileToUpdate, targetTileType);

            //Update selected tile on map
            updateTileOnMap(tileToUpdate.getCoords());
        }

        private void setTileType(Tile tileToUpdate, string targetTileType)
        {
            if (tileToUpdate.getTileType() != targetTileType)
            {
                //Don't update if trying to write Wall into 'X'
                if (!(Global.lstPointsToPathfind.Contains(tileToUpdate.getCoords()) &&
(targetTileType == "Wall")))
                {
                    //Remove old tile int dicTileTypeTiles
                    string tileCoords = tileToUpdate.getCoords();
                    string currTileType = tileToUpdate.getTileType();
                    Global.dicTileTypeTiles[currTileType].Remove(tileCoords);

                    //Add new tile in dicTileTypeTiles
                    tileToUpdate.setTileType(targetTileType);
                    Global.dicTileTypeTiles[targetTileType].Add(tileCoords);

                    string tbxOutput = "";
                    string[] tempOutput = tileToUpdate.getCoords().Split(',');
                    tbxOutput = string.Format("Tile at coordinate ({0}, {1}) set to
tileType '{2}'", tempOutput[0], tempOutput[1], targetTileType);
                    outputToTbxOutput(tbxOutput);
                }
            }
        }

        private void boundaryFill(PictureBox pbxOverlay, Tile startTile)
        {
            Queue<string> tilesToCheck = new Queue<string>();        //Holds
coordinates of tiles to be checked and the layer it belongs to
            HashSet<string> tilesSeen = new HashSet<string>();

            string startTileType = startTile.getTileType();
            Point startPointCoords = coordsStringToPoint(startTile.getCoords());
```

```
            //Enqueue Start Pos.
            tilesToCheck.Enqueue(startTile.getCoords());
            tilesSeen.Add(startTile.getCoords());

            while (tilesToCheck.Count > 0)
            {
                //Index: 0 = x, 1 = y
                string currTileCoords = tilesToCheck.Dequeue();
                Point currCoords = coordsStringToPoint(currTileCoords);

                //Look at adjacent tiles - dir = Checking direction
(up/left/down/right) = (0/1/2/3)
                for (int dir = 0; dir <= 3; dir++)
                {
                    int nextX = 0, nextY = 0;

                    switch (dir)
                    {
                     case 0:       //UP - (X, Y - 1)
                            nextX = currCoords.X;
                            nextY = currCoords.Y - 1;
                            break;
                     case 1:       //LEFT - (X - 1, Y)
                            nextX = currCoords.X - 1;
                            nextY = currCoords.Y;
                            break;
                     case 2:       //DOWN - (X, Y + 1)
                            nextX = currCoords.X;
                               nextY = currCoords.Y + 1;
                            break;
                     case 3:       //RIGHT - (X + 1, Y)
                            nextX = currCoords.X + 1;
                            nextY = currCoords.Y;
                            break;
                    }

                    //If within bounds
                    if (nextX >= 0 && nextX < Global.noOfTilesX && nextY >= 0 && nextY
< Global.noOfTilesY)
                    {
                     Tile nextTile = Global.lstMapRow[nextY][nextX];

                     //If next tile NOT seen AND equal to start tile tileType
                     if (!tilesSeen.Contains(nextTile.getCoords()) &&
                            nextTile.getTileType() == startTile.getTileType())
                     {
                            //Don't update if trying to write Wall into 'X'
                            if
(!(Global.lstPointsToPathfind.Contains(nextTile.getCoords()) &&
(Global.selectedTileType == "Wall")))
                            {
                                tilesToCheck.Enqueue(nextTile.getCoords());
                                tilesSeen.Add(nextTile.getCoords());
                            }
```

```
                    }
                   }
                }
              }

              //Update tileTypes
              foreach (string coords in tilesSeen)
              {
                Tile tileToUpdate = coordsToTile(coords);
                setTileType(tileToUpdate, Global.selectedTileType);
              }

            updateTilesOnMap(tilesSeen);

              string tbxOutput = "";
            tbxOutput = string.Format("Shape around coordinate ({0}, {1}) set to
tileType '{2}'", startPointCoords.X, startPointCoords.Y, Global.selectedTileType);
            outputToTbxOutput(tbxOutput);
        }

         #region OverlayMouseFunctions
        private void overlay_MouseEnter(object sender, EventArgs e)
        {
              PictureBox pbxOverlay = (PictureBox)sender;
              PictureBox pbxGrid = (PictureBox)pbxOverlay.Parent;
              Panel pnlScreen = (Panel)pbxGrid.Parent;

              string tbxOutput = "";
            tbxOutput = string.Format("Mouse entered {0}", pnlScreen.Name);
            outputToTbxOutput(tbxOutput);
        }

           private void overlay_MouseLeave(object sender, EventArgs e)
        {
              //Remove pbxOverlay from pbxGrid
              PictureBox pbxOverlay = (PictureBox)sender;
              PictureBox pbxGrid = (PictureBox)pbxOverlay.Parent;
              Panel pnlScreen = (Panel)pbxGrid.Parent;
              Point cursPos = pnlScreen.PointToClient(Cursor.Position);

              //Updates new tile highlight position based on last tile mouse hovered
above before leaving
              if (cursPos.X < Global.borderLength || cursPos.Y < Global.borderLength
                || cursPos.X >= pbxGrid.Width || cursPos.Y >= pbxGrid.Height)
              {
                //Makes tileHighlight offscreen
                Global.newTileHighlightX = pnlScreen.Width;
                Global.newTileHighlightY = pnlScreen.Height;
              }

              //USELESS??
              //else
              //{
              //     //Gets tile coordinates on leaving grid
```

```
            //     int offsetX = pnlGrid.Location.X + pbxGrid.Location.X;
            //     int offsetY = pnlGrid.Location.Y + pbxGrid.Location.Y;

            //    Global.newTileHighlightX = cursPos.X - offsetX;
            //    Global.newTileHighlightY = cursPos.Y - offsetY;
            //}

          replaceTileHighLight(pbxOverlay);

            string tbxOutput = "";
          tbxOutput = string.Format("Mouse left {0}", pnlScreen.Name);
          outputToTbxOutput(tbxOutput);
      }

      private void overlay_MouseMove(object sender, MouseEventArgs e)
      {
            PictureBox pbxOverlay = (PictureBox)sender;
            Point cursPos = pbxOverlay.PointToClient(Cursor.Position);

            //If not out of range
            if (!(cursPos.X < 0 || cursPos.X > pbxOverlay.Width || cursPos.Y < 0 ||
cursPos.Y > pbxOverlay.Height))
            {
              //Snap tile highlight to tile
              //Get tile no. first
              Global.newTileHighlightX = (int)(Math.Floor((decimal)cursPos.X /
Global.tileLength));
              Global.newTileHighlightY = (int)(Math.Floor((decimal)cursPos.Y /
Global.tileLength));

              //Corrects for mouse on bottom-most / right-most border drawing tile
highlights outside of pbxGrid
              if (Global.newTileHighlightX == Global.noOfTilesX)
              {
                  Global.newTileHighlightX -= 1;
              }
              if (Global.newTileHighlightY == Global.noOfTilesY)
              {
                  Global.newTileHighlightY -= 1;
              }

              //Convert to screen coordinates
              Global.newTileHighlightX *= Global.tileLength;
              Global.newTileHighlightY *= Global.tileLength;

              //Only draw new tile highlight if different from current location
              if (Global.newTileHighlightX != Global.curTileHighlightX ||
Global.newTileHighlightY != Global.curTileHighlightY)
              {
                  replaceTileHighLight(pbxOverlay);
              }

              //If any mouse button pressed
              doMouseButtonPressedEvents(e);
```

```
            }
        }

        private void overlay_MouseClick(object sender, MouseEventArgs e)
        {
            //M3 click, tabTilesOpen, map ready to edit - Boundary Fill
            if (e.Button == MouseButtons.Middle && tabCtrl.SelectedIndex == 1 &&
Global.mapReadyToEdit)
            {
              PictureBox pbxOverlay = (PictureBox)sender;

              //Get selected tile
              int tileX = Global.curTileHighlightX / Global.tileLength;
              int tileY = Global.curTileHighlightY / Global.tileLength;
              Tile tileSelected = Global.lstMapRow[tileY][tileX];

              boundaryFill(pbxOverlay, tileSelected);
            }
            else
            {
              doMouseButtonPressedEvents(e);
            }
        }

         #endregion

#endregion
```

UsefulFunctions

The code below shows a collection of methods that are used throughout multiple sections of code:

```
#region UsefulFunctions

        private void OutputMap()
        {
                string tbxOutput = string.Format("Outputting {0} x {1} map to
screen...", Global.noOfTilesX, Global.noOfTilesY);
                outputToTbxOutput(tbxOutput);

                BestFitPbx();
                DrawFullGrid();

                tbxOutput = string.Format("...done outputting {0} x {1} map to screen",
Global.noOfTilesX, Global.noOfTilesY);
                outputToTbxOutput(tbxOutput);
        }

        private Point coordsStringToPoint(string tileCoords)
        {
                string[] arrCoords = tileCoords.Split(',');
                Point coords = new Point(int.Parse(arrCoords[0]),
int.Parse(arrCoords[1]));

                return coords;
        }

        private Tile coordsToTile(string tileCoords)
        {
                Point coords = coordsStringToPoint(tileCoords);

                Tile tile = Global.lstMapRow[coords.Y][coords.X];
                return tile;
        }

        private void reorderStringList(ref List<string> lst, int indexToDelete)
        {
                //Shifting index in list downwards
                for (int i = indexToDelete; i < lst.Count() - 1; i++)
                {
                  lst[i] = lst[i + 1];
                }

                //Remove last element in list
               lst.RemoveAt(lst.Count() - 1);
        }

        private Tuple<int, string> getInitTileLength()
        {
```

```
            int testLengthX = (int)Math.Floor((decimal)(pnlGrid.Width - 1 -
Global.borderLength) / Global.noOfTilesX);
            int testLengthY = (int)Math.Floor((decimal)(pnlGrid.Height - 1 -
Global.borderLength) / Global.noOfTilesY);
            int initTileLength = 0;
            string lastAction = "";

            //Determines whether width/height/both caused tileLength to go less than
minimum
            if (testLengthX < Global.minTileLength && testLengthY >=
Global.minTileLength)
            {
              lastAction = "X";
            }
            else if (testLengthY < Global.minTileLength && testLengthX >=
Global.minTileLength)
            {
              lastAction = "Y";
            }
            else if (testLengthX < Global.minTileLength && testLengthY <
Global.minTileLength)
            {
              lastAction = "XY";
            }

            //Sets initTileLength
            if (testLengthX > testLengthY)
            {
              initTileLength = testLengthY;
         }
            else if (testLengthX < testLengthY)
            {
              initTileLength = testLengthX;
            }
            else if (testLengthX == testLengthY)
            {
              initTileLength = testLengthX;
            }

            Tuple<int, string> result = new Tuple<int, string>(initTileLength,
lastAction);
            return result;
       }

       private Panel createScreen(string screenName, Point screenLocation, Size
screenSize)
       {
            Panel pnlScreen = new Panel();
          pnlScreen.Name = screenName;
          pnlScreen.Location = screenLocation;
          pnlScreen.Size = screenSize;
          pnlScreen.BackColor = Color.White;

            Label lblScreen = new Label();
```

```
            lblScreen.Name = "lblScreen";
             lblScreen.Text = pnlScreen.Name;
             lblScreen.AutoSize = true;
             lblScreen.Location = new Point(0, 0);

               PictureBox pbxGrid = new PictureBox();
             pbxGrid.Name = "pbxGrid";
             pbxGrid.Location = new Point(Global.borderLength, Global.borderLength);
             pbxGrid.Size = new Size(pnlScreen.Width - Global.borderLength,
pnlScreen.Height - Global.borderLength);

               //NOTE: pnlScreen.Controls[0] = pbxGrid
               //     pnlScreen.Controls[1] = lblScreen

             pnlScreen.Controls.Add(pbxGrid);
             pnlScreen.Controls.Add(lblScreen);

               return pnlScreen;
       }

#endregion
```

menuBarMethods

The code below describes what would happen when options from the menu bar are clicked:

```
#region menuBarMethods
        private void menuBarNew_Click(object sender, EventArgs e)
        {
                DialogResult newProjectResult = MessageBox.Show("Start new
project?\n(All unsaved changes will be deleted)", "New Project",
MessageBoxButtons.YesNo, MessageBoxIcon.Question);
                if (newProjectResult == DialogResult.Yes)
                {
                  ResetProject();
                }
        }

        private void ResetProject()
        {
                string tbxOutput = "";
             tbxOutput = string.Format("Resetting project...");
             outputToTbxOutput(tbxOutput);

             resetTabScreen();
             resetTabTiles();
             resetPathfinding();

             OutputMap();

             tbxOutput = string.Format("...project reset");
             outputToTbxOutput(tbxOutput);
        }

        private void resetTabScreen()
        {
                string tbxOutput = "";
             tbxOutput = string.Format("Resetting tabScreen...");
             outputToTbxOutput(tbxOutput);

                //Clear all path points
                Global.lstPointsToPathfind.Clear();
             tbxOutput = string.Format("All path points cleared");
             outputToTbxOutput(tbxOutput);

                //Delete all tiles from map
                Global.lstMapRow.Clear();
             tbxOutput = string.Format("Map cleared");
             outputToTbxOutput(tbxOutput);

             deleteAllCustomScreens();

                Panel pnlAddSet = (Panel)tabScreen.Controls.Find("pnlAddSet",
false).FirstOrDefault();
```

```
            Panel pnlDefault = (Panel)tabScreen.Controls.Find("Screen0",
false).FirstOrDefault();
            Button btnAdd = (Button)pnlAddSet.Controls.Find("btnAddScreen",
false).FirstOrDefault();

            Label lblSeparator = (Label)tabScreen.Controls.Find("lblSeparator",
false).FirstOrDefault();
          pnlAddSet.Location = new Point(30, pnlDefault.Location.Y +
pnlDefault.Height);
          btnAdd.Visible = true;

            //Reinitialises map to default size
          InitialiseMap();
          tbxOutput = string.Format("Map reinitialised to default size");
          outputToTbxOutput(tbxOutput);

            //Reset width and height textbox values in tabScreen
            Panel pnlScreenSize = (Panel)tabScreen.Controls.Find("pnlScreenSize",
false).FirstOrDefault();
            TextBox tbxWidth = (TextBox)pnlScreenSize.Controls.Find("tbxWidth",
false).FirstOrDefault();
            TextBox tbxHeight = (TextBox)pnlScreenSize.Controls.Find("tbxHeight",
false).FirstOrDefault();
        tbxWidth.Text = Global.noOfTilesX.ToString();
          tbxHeight.Text = Global.noOfTilesY.ToString();

          tbxOutput = string.Format("...tabScreen reset");
          outputToTbxOutput(tbxOutput);
      }

      private void deleteAllCustomScreens()
      {
            //Deletes all screen panels but default
            for (int i = Global.lstScreens.Count() - 1; i > 0; i--)
            {
              Panel pnlToDelete = (Panel)tabScreen.Controls.Find("Screen" + i,
false).FirstOrDefault();
              tabScreen.Controls.Remove(pnlToDelete);
              Global.lstScreens.RemoveAt(i);
              pnlGrid.Controls.RemoveAt(i);
            }

            //Resets default screen panel
            Panel pnlDefault = (Panel)tabScreen.Controls.Find("Screen0",
false).FirstOrDefault();
            ComboBox cmbAlgorithm =
(ComboBox)pnlDefault.Controls.Find("cmbAlgorithm", false).FirstOrDefault();
          cmbAlgorithm.Text = "BFS";

            string tbxOutput = "";
          tbxOutput = string.Format("All custom screens deleted");
          outputToTbxOutput(tbxOutput);
      }
```

```
        private void resetTabTiles()
        {
                string tbxOutput = "";
              tbxOutput = string.Format("Resetting tabTiles...");
              outputToTbxOutput(tbxOutput);

              deleteAllCustomTileTypes();

                //Reset tileTypeSelector location
                Panel pnlWall = (Panel)tabTiles.Controls.Find("Wall",
false).FirstOrDefault();
                PictureBox pbxWall = (PictureBox)pnlWall.Controls.Find("pbxWall",
false).FirstOrDefault();
              moveTileTypeSelector(pbxWall);

              tbxOutput = string.Format("...tabTiles reset");
              outputToTbxOutput(tbxOutput);
         }

        private void deleteAllCustomTileTypes()
        {
                //Delete all custom panels
                List<Panel> pnlsToDelete = tabTiles.Controls.OfType<Panel>().ToList();
                int pnlToDeleteIndex = pnlsToDelete.Count() - 1;

              while (pnlToDeleteIndex > 2)        //Excludes first 3 panels
                {
                  Panel pnlToDelete = pnlsToDelete[pnlToDeleteIndex];
                  tabTiles.Controls.Remove(pnlToDelete);
                  pnlToDeleteIndex--;
           }

                //Moves pnlAddSet to starting position and makes the '+' button visible
and the 'Set' button invisible
                Panel pnlAddSet = pnlsToDelete[2];
                Button btnAdd = (Button)pnlAddSet.Controls.Find("btnAddTileType",
false).FirstOrDefault();
                Button btnSet = (Button)pnlAddSet.Controls.Find("btnSetTileTypes",
false).FirstOrDefault();

                Label lblSeparator = (Label)tabTiles.Controls.Find("lblSeparator",
false).FirstOrDefault();
              pnlAddSet.Location = new Point(30, lblSeparator.Location.Y + 30);
              btnAdd.Visible = true;
              btnSet.Visible = false;

              pnlsToDelete.Clear();

                //Reset list of dictionary keys
                Global.lstDicTileTypeKeys.Clear();
                Global.lstDicTileTypeKeys.Add("Empty");
                Global.lstDicTileTypeKeys.Add("Wall");

                //Reset tile type info dictionary
```

```
            Global.dicTileTypeInfo.Clear();
            Global.dicTileTypeInfo.Add("Empty", new Tuple<Color, int>(Color.White,
1));
            Global.dicTileTypeInfo.Add("Wall", new Tuple<Color, int>(Color.Black,
-1));

            //Reset tiles of certain tileType dictionary
            Global.dicTileTypeTiles.Clear();
            Global.dicTileTypeTiles.Add("Empty", new HashSet<string>());
            Global.dicTileTypeTiles.Add("Wall", new HashSet<string>());

            string tbxOutput = "";
          tbxOutput = string.Format("All custom tileTypes deleted");
          outputToTbxOutput(tbxOutput);
      }

      private void menuBarImportImage_Click(object sender, EventArgs e)
      {
            //Checkers - Exceeds max tileTypes?
            OpenFileDialog dlgOpen = new OpenFileDialog();
            DialogResult result = dlgOpen.ShowDialog();
            if (result == DialogResult.OK)
            {
              string tbxOutput = "";
              tbxOutput = string.Format("Attempting to import image...");
              outputToTbxOutput(tbxOutput);

              string fileDir = dlgOpen.FileName;
              bool isImage = false;

              //Check if file is an image
              try
              {
                  Bitmap bmpImport = (Bitmap)Image.FromFile(fileDir);
                  isImage = true;
              }
              catch
              {
                  tbxOutput = string.Format("...could not import image");
                  outputToTbxOutput(tbxOutput);

                  MessageBox.Show("Error: Can't import " +
fileDir.Substring(fileDir.LastIndexOf('.')));
              }

              //Proceed only if imported file is an image
              if (isImage)
              {
                  Bitmap bmpImport = (Bitmap)Image.FromFile(fileDir);

                  tbxOutput = string.Format("Testing if image fits into all
screens...");
                  outputToTbxOutput(tbxOutput);
```

```
                    //Trial fit to test if image will fit into current number of
screens
                int oldNoOfTilesX = Global.noOfTilesX;
                    int oldNoOfTilesY = Global.noOfTilesY;
                    Global.noOfTilesX = bmpImport.Width;
                    Global.noOfTilesY = bmpImport.Height;

                    Tuple<int, string> tileLengthOptions = trialFitScreen();

                    Global.noOfTilesX = oldNoOfTilesX;
                    Global.noOfTilesY = oldNoOfTilesY;
                    int testTileLength = tileLengthOptions.Item1;

                    //If image is not too large
                    if (testTileLength != -1)
                    {
                     tbxOutput = string.Format("...image fits");
                        outputToTbxOutput(tbxOutput);

                     //Goes through bitmap and creates list of unique colours
                     List<Color> lstColour = new List<Color>();
                     Rectangle rect = new Rectangle(0, 0, bmpImport.Width,
bmpImport.Height);
                     BitmapData bmpData = bmpImport.LockBits(rect,
ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);
                     fillColourList(ref lstColour, ref bmpData);

                     tbxOutput = string.Format("Testing if there are enough tileType
slots for each unique colour in image...");
                        outputToTbxOutput(tbxOutput);

                     //A tileType will be created for each unique colour present in
the bmp
                     //Check to see if there are enough spaces for these tileTypes
                     if (lstColour.Count() <= Global.maxTileTypes)
                     {
                            tbxOutput = string.Format("...there are enough tileType
slots");
                             outputToTbxOutput(tbxOutput);

                            //Notify user that any unsaved changes will be erased
                            DialogResult newProjectResult = MessageBox.Show("All
unsaved changes will be deleted. Continue?", "Import bitmap", MessageBoxButtons.YesNo,
MessageBoxIcon.Question);
                            if (newProjectResult == DialogResult.Yes)
                            {
                                Global.noOfTilesX = bmpImport.Width;
                               Global.noOfTilesY = bmpImport.Height;
                                  Global.tileLength = testTileLength;
                                prepareImport();

                                createTileTypes(ref lstColour);
                                  outputBmpToMap(ref bmpData);
                                bmpImport.UnlockBits(bmpData);
```

```
                                placeScreens();
                                DrawFullGrid();

                                tbxOutput = string.Format("...image successfully
imported");
                            outputToTbxOutput(tbxOutput);
                        }
                        else
                        {
                                tbxOutput = string.Format("...image importing
cancelled");
                            outputToTbxOutput(tbxOutput);
                        }
                    }
                    else
                    {
                            bmpImport.UnlockBits(bmpData);

                            tbxOutput = string.Format("...too many colours");
                            outputToTbxOutput(tbxOutput);
                            tbxOutput = string.Format("...could not import image");
                            outputToTbxOutput(tbxOutput);

                            MessageBox.Show("Error: Too many colours in image.
Maximum: " + Global.maxTileTypes + " excluding black and white. Your image has " +
lstColour.Count());
                    }
                    }
                    else
                    {
                        tbxOutput = string.Format("...image too large");
                            outputToTbxOutput(tbxOutput);
                        tbxOutput = string.Format("...could not import image");
                            outputToTbxOutput(tbxOutput);

                        MessageBox.Show("Error: Image too large. Maximum size for " +
Global.lstScreens.Count() + " " + ((Global.lstScreens.Count() > 1) ? "screens" :
"screen") + ": " +
                                                +Global.maxNoOfTilesX + " x " +
Global.maxNoOfTilesY + ". Your image is " + bmpImport.Width + " x " +
bmpImport.Height);
                    }
                }
            }
        }

        private void fillColourList(ref List<Color> lstColour, ref BitmapData bmpData)
        {
                unsafe
                {
                  //Get address of first line
                  byte* ptrStart = (byte*)bmpData.Scan0;
                  int Bpp = 4;
```

```
                byte A, R, G, B;

                if (BitConverter.IsLittleEndian)
                {
                    for (int row = 0; row < bmpData.Height; row++)
                    {
                     byte* ptrRow = ptrStart + (row * (Bpp * bmpData.Width));

                     for (int col = 0; col < bmpData.Width; col++)
                     {
                            int bmpX = col * Bpp;

                            B = ptrRow[bmpX];
                            G = ptrRow[bmpX + 1];
                            R = ptrRow[bmpX + 2];
                            A = ptrRow[bmpX + 3];

                            Color pixColour = Color.FromArgb(A, R, G, B);
                            if (!lstColour.Contains(pixColour))
                            {
                                   //Add if not default tileType colour
                                   if (!(pixColour.ToArgb() == Color.White.ToArgb() ||
pixColour.ToArgb() == Color.Black.ToArgb()))
                                      {
                                        lstColour.Add(pixColour);
                                      }
                            }
                     }
                    }
                }
                //For big endian
                else
                {
                    for (int row = 0; row < bmpData.Height; row++)
                    {
                     byte* ptrRow = ptrStart + (row * (Bpp * bmpData.Width));

                     for (int col = 0; col < bmpData.Width; col++)
                     {
                            Color tileColour = Global.lstMapRow[row][col].getColour();

                            int bmpX = col * Bpp;

                            A = ptrRow[bmpX];
                            R = ptrRow[bmpX + 1];
                            G = ptrRow[bmpX + 2];
                            B = ptrRow[bmpX + 3];

                            Color pixColour = Color.FromArgb(A, R, G, B);
                            if (!lstColour.Contains(pixColour))
                            {
                                   //Add if not default tileType colour
                                   if (!(pixColour.ToArgb() == Color.White.ToArgb() ||
pixColour.ToArgb() == Color.Black.ToArgb()))
```

```
                                {
                                  lstColour.Add(pixColour);
                                }
                            }
                        }
                    }
                }
            }

        private void prepareImport()
        {
            string tbxOutput = "";
            tbxOutput = string.Format("Preparing to import image...");
            outputToTbxOutput(tbxOutput);

            //Clear all path points
            Global.lstPointsToPathfind.Clear();
            tbxOutput = string.Format("All path points cleared");
            outputToTbxOutput(tbxOutput);

            //Reset tile map
            Global.lstMapRow.Clear();
            tbxOutput = string.Format("Map cleared");
            outputToTbxOutput(tbxOutput);

            //Rebuilding map - Need to change Global.noOfTilesXY temporarily
            int noOfTilesX = Global.noOfTilesX;
            int noOfTilesY = Global.noOfTilesY;
            Global.noOfTilesX = 0;
            Global.noOfTilesY = 0;

            rebuildMap(noOfTilesX, noOfTilesY);

            Global.noOfTilesX = noOfTilesX;
            Global.noOfTilesY = noOfTilesY;

            //Clear empty hashSet to prevent excess tiles being stored later on
            Global.dicTileTypeTiles["Empty"].Clear();

            //Reset width and height textbox values in tabScreen
            Panel pnlScreenSize = (Panel)tabScreen.Controls.Find("pnlScreenSize",
false).FirstOrDefault();
            TextBox tbxWidth = (TextBox)pnlScreenSize.Controls.Find("tbxWidth",
false).FirstOrDefault();
            TextBox tbxHeight = (TextBox)pnlScreenSize.Controls.Find("tbxHeight",
false).FirstOrDefault();
            tbxWidth.Text = Global.noOfTilesX.ToString();
            tbxHeight.Text = Global.noOfTilesY.ToString();

            resetTabTiles();
            resetPathfinding();

            tbxOutput = string.Format("...preparations complete");
```

```
            outputToTbxOutput(tbxOutput);
        }

        private void createTileTypes(ref List<Color> lstColour)
        {
                Panel pnlAddSet = (Panel)tabTiles.Controls.Find("pnlAddSet",
false).FirstOrDefault();
                Button btnAdd = (Button)pnlAddSet.Controls.Find("btnAddTileType",
false).First();

                //Create tileTypes for each unique colour
                for (int i = 0; i < lstColour.Count(); i++)
                {
                  Color tileColour = lstColour[i];

                  //Add tileType panel
                  btnAddTileType_Click(btnAdd, EventArgs.Empty);

                  //Change colour of tileType in panel that was just added
                  Panel pnlTileType = (Panel)tabTiles.Controls[tabTiles.Controls.Count -
1];
                  PictureBox pbxTileTypeColour =
(PictureBox)pnlTileType.Controls.Find("pbxColour", false).FirstOrDefault();
                  pbxTileTypeColour.Tag = tileColour;           //Tag colour on pbx so
we can refer to it when updating
                  drawTileTypePbx(pbxTileTypeColour, tileColour);

                  //Update dictionary entries (Created in btnAddTileType_Click)
                  string newTileTypeName = tileColour.ToString();

                  //Recreate dicTileTypeTiles entry
                  Global.dicTileTypeTiles.Remove(pnlTileType.Name);
                  Global.dicTileTypeTiles.Add(newTileTypeName, new HashSet<string>());

                  //Recreate dicTileTypeInfo entry (Note: Weight = 1)
                  Global.dicTileTypeInfo.Remove(pnlTileType.Name);
                  Global.dicTileTypeInfo.Add(newTileTypeName, new Tuple<Color,
int>(tileColour, 1));

                  //Update list of keys
                  int indexToChange =
Global.lstDicTileTypeKeys.IndexOf(pnlTileType.Name);
                  Global.lstDicTileTypeKeys[indexToChange] = newTileTypeName;

                  //Note: tileType name = Colour of tileType (argb), since on import,
there are no two tileTypes with same colour
                  pnlTileType.Name = newTileTypeName;
                }
        }

        private void outputBmpToMap(ref BitmapData bmpData)
        {
                unsafe
                {
```

```
//Get address of first line
byte* ptrStart = (byte*)bmpData.Scan0;
int Bpp = 4;
byte A, R, G, B;

if (BitConverter.IsLittleEndian)
{
    for (int row = 0; row < Global.noOfTilesY; row++)
    {
     byte* ptrRow = ptrStart + (row * (Bpp * Global.noOfTilesX));

     for (int col = 0; col < Global.noOfTilesX; col++)
     {
            int bmpX = col * Bpp;

            B = ptrRow[bmpX];
            G = ptrRow[bmpX + 1];
            R = ptrRow[bmpX + 2];
            A = ptrRow[bmpX + 3];

            Color pixColour = Color.FromArgb(A, R, G, B);
            string tileTypeName = "";

            //Add tiles to hashSets
          if (pixColour.ToArgb() == Color.White.ToArgb())
            {
                   tileTypeName = "Empty";
            }
            else if (pixColour.ToArgb() == Color.Black.ToArgb())
            {
                   tileTypeName = "Wall";
            }
            else
            {
                   tileTypeName = pixColour.ToString();
             }

            //Edit map data with new colour
            Global.lstMapRow[row][col].setTileType(tileTypeName);

            //Add new coordinate to hashset
            string coords = col.ToString() + "," + row.ToString();
            Global.dicTileTypeTiles[tileTypeName].Add(coords);
     }
    }
}
//For big endian
else
{
    for (int row = 0; row < Global.noOfTilesY; row++)
    {
     byte* ptrRow = ptrStart + (row * (Bpp * Global.noOfTilesX));

     for (int col = 0; col < Global.noOfTilesX; col++)
```

```
                {
                        Color tileColour = Global.lstMapRow[row][col].getColour();

                        int bmpX = col * Bpp;

                        A = ptrRow[bmpX];
                        R = ptrRow[bmpX + 1];
                        G = ptrRow[bmpX + 2];
                        B = ptrRow[bmpX + 3];

                        Color pixColour = Color.FromArgb(A, R, G, B);
                        string tileTypeName = "";

                        //Add tiles to hashSets
                        if (pixColour.ToArgb() == Color.White.ToArgb())
                        {
                                tileTypeName = "Empty";
                        }
                        else if (pixColour.ToArgb() == Color.Black.ToArgb())
                        {
                                tileTypeName = "Wall";
                        }
                        else
                         {
                                tileTypeName = pixColour.ToString();
                        }

                        //Edit map data with new colour
                        Global.lstMapRow[row][col].setTileType(tileTypeName);

                    //Add new coordinate to hashset
                        string coords = col.ToString() + "," + row.ToString();
                        Global.dicTileTypeTiles[tileTypeName].Add(coords);
                }
                }
            }
        }
}

private void menuBarExportBmp_Click(object sender, EventArgs e)
{
        SaveFileDialog dlgSave = new SaveFileDialog();
     dlgSave.DefaultExt = ".bmp";
   dlgSave.ShowDialog();
        string fileDir = "";
     fileDir = dlgSave.FileName;

        //If name is valid
        if (fileDir != "")
        {
          Bitmap bmp = new Bitmap(Global.noOfTilesX, Global.noOfTilesY);

          //Lock bmp bits to sys mem
          Rectangle rect = new Rectangle(0, 0, bmp.Width, bmp.Height);
```

```
                BitmapData bmpData = bmp.LockBits(rect, ImageLockMode.ReadWrite,
PixelFormat.Format32bppArgb);
                int Bpp = 4;

                unsafe
                {
                    //Get address of first line
                    byte* ptrStart = (byte*)bmpData.Scan0;

                    //Fill bmp
                    if (BitConverter.IsLittleEndian)
                    {
                     for (int row = 0; row < Global.noOfTilesY; row++)
                     {
                            byte* ptrRow = ptrStart + (row * (Bpp *
Global.noOfTilesX));

                            for (int col = 0; col < Global.noOfTilesX; col++)
                            {
                                Color tileColour =
Global.lstMapRow[row][col].getColour();

                                int bmpX = col * Bpp;

                                ptrRow[bmpX] = tileColour.B;
                                ptrRow[bmpX + 1] = tileColour.G;
                                ptrRow[bmpX + 2] = tileColour.R;
                                ptrRow[bmpX + 3] = tileColour.A;
                            }
                     }
                    }
                    //For big endian
                    else
                    {
                     for (int row = 0; row < Global.noOfTilesY; row++)
                        {
                            byte* ptrRow = ptrStart + (row * (Bpp *
Global.noOfTilesX));

                            for (int col = 0; col < Global.noOfTilesX; col++)
                            {
                                Color tileColour =
Global.lstMapRow[row][col].getColour();

                                int bmpX = col * Bpp;

                                ptrRow[bmpX] = tileColour.A;
                                ptrRow[bmpX + 1] = tileColour.R;
                                ptrRow[bmpX + 2] = tileColour.G;
                                ptrRow[bmpX + 3] = tileColour.B;
                            }
                     }
                    }
```

```
                    bmp.UnlockBits(bmpData);
                }

                bmp.Save(fileDir);

                string tbxOutput = "";
                tbxOutput = string.Format("Map exported as bitmap");
                outputToTbxOutput(tbxOutput);
            }
        }

        private void menuBarExit_Click(object sender, EventArgs e)
        {
            DialogResult exitResult = MessageBox.Show("Exit the program?", "Exit",
MessageBoxButtons.YesNo, MessageBoxIcon.Question);
            if (exitResult == DialogResult.Yes)
            {
                Application.Exit();
            }
        }
#endregion
```

Pathfinding

The code below describes how pathfinding will work. It also includes implementations of the pathfinding algorithms described in the design section, which can be found in the “PathfindingAlgorithms” region. In particular, the Dijkstra and A* algorithms will feature use of the classes PriorityListDijkstra and PriorityListAStar. All pathfinding algorithms will use stacks when it comes to backtracking the shortest path from the destination node to the source node:

```
#region Pathfinding

        //SPLIT INTO MORE METHODS, MAKE EASIER TO READ
        private void btnStartPathfind_Click(object sender, EventArgs e)
        {
                Button btnStartPathfind = (Button)sender;

                //Starting pathfinding
                if (btnStartPathfind.Text == "Start pathfinding")
                {
                  string tbxOutput = "";
                  tbxOutput = string.Format("");
                  outputToTbxOutput(tbxOutput);
                  tbxOutput = string.Format("Pathfinding started...");
                  outputToTbxOutput(tbxOutput);

                  //Multiple destinations
                  if (Global.lstPointsToPathfind.Count > 2)
                  {
                      tbxOutput = string.Format("...too many path points");
                      outputToTbxOutput(tbxOutput);

                      MessageBox.Show("Multiple destinations (with Minimum spanning
tree, Complete Graphs) not yet available", "Coming soon!", MessageBoxButtons.OK,
MessageBoxIcon.Information);
                  }
                  //Single destination
                  else if (Global.lstPointsToPathfind.Count == 2)
                  {
                      //Switch to tabOutput screen
                      tabCtrl.SelectedIndex = 2;
                      tabOutput.Refresh();

                      //Pathfind for each overlay
                      for (int i = 0; i < pnlGrid.Controls.Count; i++)
                      {
                       Panel pnlScreen = (Panel)pnlGrid.Controls[i];
                       PictureBox pbxGrid = (PictureBox)pnlScreen.Controls[0];
                       PictureBox pbxOverlay = (PictureBox)pbxGrid.Controls[0];

                       Panel pnlScreenOptions =
(Panel)tabScreen.Controls.Find(pnlScreen.Name, false).FirstOrDefault();
```

```
                    ComboBox cmbAlgorithm =
(ComboBox)pnlScreenOptions.Controls.Find("cmbAlgorithm", false).FirstOrDefault();

                    //Find chosen algorithm and its options then start pathfinding
                       startPathfinding(pnlScreenOptions, pbxOverlay, cmbAlgorithm);

                    //Stops pathfinding if no path
                    if (btnStartPathfind.Text == "Reset pathfinding")
                         {
                           tbxOutput = string.Format("No path found. Pathfinding
stopped");
                            outputToTbxOutput(tbxOutput);

                           i = pnlGrid.Controls.Count - 1;
                     }

                    tabOutput.Refresh();
                   }

                   btnStartPathfind.Text = "Reset pathfinding";

                   Global.mapReadyToEdit = false;
                   Global.allowEditPoints = false;
                   Global.donePathfinding = true;

                   tbxOutput = string.Format("...pathfinding ended");
                   outputToTbxOutput(tbxOutput);
               }
               //Too little path points
               else
               {
                   tbxOutput = string.Format("...too little path points on map");
                   outputToTbxOutput(tbxOutput);

                   MessageBox.Show("No start and end point defined", "Not enough path
points", MessageBoxButtons.OK, MessageBoxIcon.Information);
               }
           }
           //Resetting pathfinding
           else if (btnStartPathfind.Text == "Reset pathfinding")
           {
             foreach (Panel pnlScreen in pnlGrid.Controls)
             {
                 PictureBox pbxGrid = (PictureBox)pnlScreen.Controls[0];
                 initPbxGridOverlay(pbxGrid);
             }

             resetPathfinding();
           }

           //SAVE BELOW FOR CONSTRUCTING COMPLETE GRAPHS
           //if (Global.lstPointsToPathfind.Count > 2)
           //{
           //     //Finding paths to all points
```

```
            //     for (int i = 0; i < Global.lstPointsToPathfind.Count; i++)
            //     {
            //     for (int j = i + 1; j < Global.lstPointsToPathfind.Count; j++)
            //     {
            //            //NEED TO REFRESH BITMAP FOR OLD GREY TILES
            //            pathfindBFS(pbxOverlay, Global.lstPointsToPathfind[i],
Global.lstPointsToPathfind[j]);
            //     }
            //     }
            //}
            //else if (Global.lstPointsToPathfind.Count == 2)
            //{
            //    pathfindBFS(pbxOverlay, Global.lstPointsToPathfind[0],
Global.lstPointsToPathfind[1]);
            //}
            //else
            //{
            //    MessageBox.Show("No start and end point defined", "Not enough path
points", MessageBoxButtons.OK, MessageBoxIcon.Information);
            //}
      }

      private void startPathfinding(Panel pnlScreenOptions, PictureBox pbxOverlay,
ComboBox cmbAlgorithm)
      {
            if (cmbAlgorithm.Text == "BFS")
            {
              pathfindBFS(pbxOverlay, Global.lstPointsToPathfind[0],
Global.lstPointsToPathfind[1]);
            }
            else if (cmbAlgorithm.Text == "Dijkstra")
            {
              pathfindDijkstra(pbxOverlay, Global.lstPointsToPathfind[0],
Global.lstPointsToPathfind[1]);
            }
            else if (cmbAlgorithm.Text == "A*")
            {
              ComboBox cmbHeuristic =
(ComboBox)pnlScreenOptions.Controls.Find("cmbHeuristic", false).FirstOrDefault();
              string heuristic = cmbHeuristic.Text.ToUpper();
              pathfindAStar(pbxOverlay, Global.lstPointsToPathfind[0],
Global.lstPointsToPathfind[1], heuristic);
            }
      }

      private void resetPathfinding()
      {
            Button btnStartPathfind = (Button)this.Controls.Find("btnStartPathfind",
false).FirstOrDefault();
           btnStartPathfind.Text = "Start pathfinding";

            Panel pnlAddPoints = (Panel)tabScreen.Controls.Find("pnlAddPoints",
false).FirstOrDefault();
```

```
             Label lblAddPoints = (Label)pnlAddPoints.Controls.Find("lblAddPoints",
false).FirstOrDefault();
            lblAddPoints.ForeColor = Color.Black;
            lblAddPoints.Text = "Path points edit mode: OFF";
              Global.allowEditPoints = false;

              Global.mapReadyToEdit = true;
              Global.donePathfinding = false;

              string tbxOutput = "";
            tbxOutput = string.Format("Pathfinding reset");
            outputToTbxOutput(tbxOutput);
        }

        private void clearAllPathPoints()
        {
              Panel pnlScreen = (Panel)pnlGrid.Controls[0];
              PictureBox pbxGrid = (PictureBox)pnlScreen.Controls[0];
              PictureBox pbxOverlay = (PictureBox)pbxGrid.Controls[0];
              Bitmap bmpOverlay = (Bitmap)pbxOverlay.Image;

              foreach (string strCoords in Global.lstPointsToPathfind)
              {
                Point coords = coordsStringToPoint(strCoords);

                //Erases X on all overlays
                foreach (Panel screen in pnlGrid.Controls)
                {
                    PictureBox grid = (PictureBox)screen.Controls[0];
                    PictureBox overlay = (PictureBox)grid.Controls[0];
                    Bitmap bmp = (Bitmap)overlay.Image;

                     //Erases cross
                    Graphics gfx = Graphics.FromImage(bmp);
                    gfx.CompositingMode =
System.Drawing.Drawing2D.CompositingMode.SourceCopy;   //Replaces pixels rather than
blend
                    gfx.FillRectangle(Brushes.Transparent, coords.X *
Global.tileLength + 1, coords.Y * Global.tileLength + 1, Global.tileLength - 1,
Global.tileLength - 1);
                }
              }
            pnlGrid.Refresh();

              Global.lstPointsToPathfind.Clear();

              string tbxOutput = "";
            tbxOutput = string.Format("All path points cleared");
            outputToTbxOutput(tbxOutput);
        }



         #region PathfindingAlgorithms
```

```
        //NOTE: BFS can only handle black and white - If coloured, treat as impassable
(wall)
        //(Colour key: Yellow = Working, Grey = Seen)
        private void pathfindBFS(PictureBox pbxOverlay, string startPoint, string
endPoint)
        {
                /* BFS Algorithm:
                *
                * Add start point to queue tilesToCheck
                * Add start point with layerNo = 0 to dictionary tilesSeen
                *
                * While tilesToCheck NOT empty
                *       currTile = tilesToCheck.Dequeue
                *       Mark currTile as 'Working' and display on overlay
                *
                *       For each nextTile adjacent to currTile
                *               If nextTile within bounds, not marked as seen, and has
'Empty' tileType
               *               If nextTile = endPoint
                *                       Clear tilesToCheck
                *                       pathFound = True
                *               Else
                *                    tilesToCheck.Enqueue(nextTile)
                *                       Mark nextTile as 'Seen' and display on overlay
                *
                *               Add nextTile with its layerNo into tilesSeen
                *
                *       Mark currTile as 'Seen' and display on overlay
                *
                * If pathFound
                *       Trace Shortest Path
                * Else
                *       Output "Could not find path to destination"
                */

                PictureBox pbxGrid = (PictureBox)pbxOverlay.Parent;
                Panel pnlScreen = (Panel)pbxGrid.Parent;
                string tbxOutput = "";
              tbxOutput = string.Format("Pathfinding through {0} with BFS...",
pnlScreen.Name);
              outputToTbxOutput(tbxOutput);

                 Stopwatch stopWatch = new Stopwatch();
              stopWatch.Start();

                Queue<string> tilesToCheck = new Queue<string>();        //Holds
coordinates of tiles to be checked
                Dictionary<string, int> tilesSeen = new Dictionary<string, int>();
//Holds tiles seen and layer no.
                bool pathFound = false;

                //Enqueue Start Pos
                Point startPointCoords = coordsStringToPoint(startPoint);
```

```
            tilesToCheck.Enqueue(startPoint);
            tilesSeen.Add(startPoint, 0);

              while (tilesToCheck.Count > 0)
              {
                string currTileCoords = tilesToCheck.Dequeue();

                //Index: 0 = x, 1 = y
                Point currCoords = coordsStringToPoint(currTileCoords);

                //Recolour current tile status
                setOverlayTileStatus(pbxOverlay, currTileCoords, Color.LightYellow);
//Yellow - Working tile

                //Look at adjacent tiles - dir = Checking direction
(up/left/down/right) = (0/1/2/3)
                for (int dir = 0; dir <= 3; dir++)
                {
                    int nextX = 0, nextY = 0;

                    switch (dir)
                    {
                     case 0:       //UP - (X, Y - 1)
                            nextX = currCoords.X;
                            nextY = currCoords.Y - 1;
                            break;
                     case 1:       //LEFT - (X - 1, Y)
                            nextX = currCoords.X - 1;
                           nextY = currCoords.Y;
                            break;
                     case 2:       //DOWN - (X, Y + 1)
                            nextX = currCoords.X;
                            nextY = currCoords.Y + 1;
                            break;
                     case 3:       //RIGHT - (X + 1, Y)
                            nextX = currCoords.X + 1;
                            nextY = currCoords.Y;
                            break;
                    }

                    //If within bounds
                    if (nextX >= 0 && nextX < Global.noOfTilesX && nextY >= 0 && nextY
< Global.noOfTilesY)
                    {
                     Tile nextTile = Global.lstMapRow[nextY][nextX];

                        //If next tile NOT seen AND empty
                     if (!tilesSeen.ContainsKey(nextTile.getCoords()) &&
nextTile.getTileType() == "Empty")
                     {
                            //If next tile is DESTINATION
                            if (nextTile.getCoords() == endPoint)
                            {
                                   //End Search
```

```
                                tilesToCheck.Clear();
                                   dir = 3;
                                pathFound = true;
                            }
                            else
                            {
                                   //Enqueue next tile, mark as seen
                                tilesToCheck.Enqueue(nextTile.getCoords());

                                setOverlayTileStatus(pbxOverlay, nextTile.getCoords(),
Color.Orange);      //Orange - Looking
                                revertTileOverlayColour(ref pbxOverlay,
nextTile.getCoords());
                            }

                             tilesSeen.Add(nextTile.getCoords(),
tilesSeen[currTileCoords] + 1);
                        }
                       }
                  }

                  revertTileOverlayColour(ref pbxOverlay, currTileCoords);
                }

              stopWatch.Stop();

                if (pathFound)
                {
                  tbxOutput = string.Format("Path found!");
                  outputToTbxOutput(tbxOutput);

                     BFSOutputPath(pbxOverlay, ref tilesSeen, endPoint);
                  tbxOutput = string.Format("Shortest path output to screen");
                  outputToTbxOutput(tbxOutput);
                }
                else
                {
                  //How to solve for multiple nodes? - To SOME destinations
                  MessageBox.Show("Could not find path to destination", "No path",
MessageBoxButtons.OK, MessageBoxIcon.Information);

                  //Change 'start pathfinding' button to 'reset'
                  Button btnStartPathfind =
(Button)this.Controls.Find("btnStartPathfind", false).FirstOrDefault();
                  btnStartPathfind.Text = "Reset pathfinding";
                }

              tbxOutput = string.Format("...pathfinding complete in {0} ms",
stopWatch.ElapsedMilliseconds);
              outputToTbxOutput(tbxOutput);
        }

        private bool isNextTilePath(PictureBox pbxOverlay, string tileCoords)
        {
```

```
            bool isTilePath = true;

            Color tileColour = getOverlayTileStatus(pbxOverlay, tileCoords);
            if (tileColour.ToArgb() != Global.pathColour.ToArgb())
            {
              isTilePath = false;
            }

            return isTilePath;
        }

        //Trace shortest path from end to start -Note: There may be multiple shortest
paths, this is one of them
        private void BFSOutputPath(PictureBox pbxOverlay, ref Dictionary<string, int>
tilesSeen, string endPoint)
        {
            /* Tracing shortest path [BFS]:
            *
            * layerNo = endPoint's layerNo
            * Push endPoint into shortestPath Stack
            * currTile = endPoint
            *
            * While layerNo > 0
            *      For each nextTile adjacent to currTile
            *             If dictionary tilesSeen contains currTile as key
            *             If layerNo of currTile = layerNo - 1
            *                    Push currTile into shortestPath
            *
           * While shortestPath NOT empty
            *      pathTile = shortestPath.Pop
            *      Mark pathTile as 'Path' and display on overlay
            */

            Stack<string> shortestPath = new Stack<string>();
            int layerNo = tilesSeen[endPoint];
            string currCoords = endPoint;
            int nextX = 0, nextY = 0;

          shortestPath.Push(endPoint);

            while (layerNo > 0)
            {
              Point coords = coordsStringToPoint(currCoords);

              //Look at adjacent tiles - dir = Checking direction
(up/left/down/right) = (0/1/2/3)
              for (int dir = 0; dir <= 3; dir++)
              {
                  switch (dir)
                  {
                   case 0:       //UP - (X, Y - 1)
                          nextX = coords.X;
                          nextY = coords.Y - 1;
                          break;
```

```
                    case 1:       //LEFT - (X - 1, Y)
                        nextX = coords.X - 1;
                        nextY = coords.Y;
                        break;
                    case 2:       //DOWN - (X, Y + 1)
                        nextX = coords.X;
                        nextY = coords.Y + 1;
                        break;
                    case 3:       //RIGHT - (X + 1, Y)
                        nextX = coords.X + 1;
                        nextY = coords.Y;
                        break;
                }

                //If adjacent tile is on last layer, add to stack. Continue
searching for previous layer
                currCoords = nextX + "," + nextY;
                if (tilesSeen.ContainsKey(currCoords))
                {
                 if (tilesSeen[currCoords] == layerNo - 1)
                 {
                        shortestPath.Push(currCoords);
                        dir = 3;
                        layerNo--;
                 }
                }
              }
            }

            //Output shortest path
            while (shortestPath.Count > 0)
            {
              string pathCoords = shortestPath.Pop();
              setOverlayTileStatus(pbxOverlay, pathCoords, Color.DarkRed);

              //Retains 'X's on overlay
              if (tilesSeen[pathCoords] == 0 || pathCoords == endPoint)
              {
                  drawXOnOverlay((Bitmap)pbxOverlay.Image, pathCoords);
                  pbxOverlay.Refresh();
              }
            }
      }

      //BEWARE OF OBTAINING DISTANCES THROUGH VERY LARGE WEIGHTED NODES - OVERFLOW
      private void pathfindDijkstra(PictureBox pbxOverlay, string startPoint, string
endPoint)
      {
            /* Dijkstra's Algorithm:
            *
            * Add start point (key) with tentative distance of 0 (val) to dictionary
tilesToCheck
            * Add start point (key) with tentative distance of 0 and previous
connected tile of "" (val) dictionary tilesSeen
```

```
             *
             * While tilesToCheck NOT empty
             *      currTile = Node with least tentative distance
            *       Remove this node from dictionary tilesToCheck
             *      Mark currTile as 'Working' and display on overlay
             *
             *      For each nextTile adjacent to currTile
             *             If nextTile within bounds, not marked as seen
             *             If nextTile = endPoint
             *                    Clear tilesToCheck
             *                    pathFound = True
             *             Else
             *                    nextTileDist = currTile's tentDist + nextTile's
weight
             *                  tilesToCheck.Add(nextTileCoords, nextTileDist);
             *                    Mark nextTile as 'Seen' and display on overlay
             *
             *               Add nextTile with nextTileDist and currTile (as previous
connected tile) into tilesSeen
             *
             *             Else If tile previously seen
             *             Mark nextTile as 'Updating' and display on overlay
            *              Compare nextTile's current tentative distance with
nextTileDist
             *             Update nextTile's tentative distance if nextTileDist is
less than the current value
             *             Mark nextTile as 'Seen' and display on overlay
             *
             *      Mark currTile as 'Seen' and display on overlay
             *
             * If pathFound
             *      Trace Shortest Path
             * Else
             *      Output "Could not find path to destination"
             */

              PictureBox pbxGrid = (PictureBox)pbxOverlay.Parent;
              Panel pnlScreen = (Panel)pbxGrid.Parent;
              string tbxOutput = "";
            tbxOutput = string.Format("Pathfinding through {0} with Dijkstra...",
pnlScreen.Name);
            outputToTbxOutput(tbxOutput);

              Stopwatch stopWatch = new Stopwatch();
            stopWatch.Start();

              //Holds coordinates of tiles to check (key) and tentative distance (val)
              PriorityListDijkstra tilesToCheck = new PriorityListDijkstra();

              //Holds tilesSeen (key) and (tentative distance, previous connected
tile) (val)
              Dictionary<string, Tuple<int, string>> tilesSeen = new
Dictionary<string, Tuple<int, string>>();
              bool pathFound = false;
```

```
            //Enqueue Start Pos
            tilesToCheck.Add(new Tuple<int, string>(0, startPoint));
            tilesSeen.Add(startPoint, new Tuple<int, string>(0, ""));

            while (tilesToCheck.Count() > 0)
            {
                //Dequeue tile
                Tuple<int, string> currTileInfo = tilesToCheck.dequeue();
                int currTileDist = currTileInfo.Item1;
                string currTileCoords = currTileInfo.Item2;

                //Index: 0 = x, 1 = y
                Point currCoords = coordsStringToPoint(currTileCoords);

                //Recolour current tile status
                setOverlayTileStatus(pbxOverlay, currTileCoords, Color.LightYellow);
//Yellow - Working tile

                //Look at adjacent tiles - dir = Checking direction
(up/left/down/right) = (0/1/2/3)
                for (int dir = 0; dir <= 3; dir++)
                {
                    int nextX = 0, nextY = 0;

                    switch (dir)
                    {
                     case 0:       //UP - (X, Y - 1)
                            nextX = currCoords.X;
                            nextY = currCoords.Y - 1;
                        break;
                     case 1:       //LEFT - (X - 1, Y)
                            nextX = currCoords.X - 1;
                            nextY = currCoords.Y;
                            break;
                     case 2:       //DOWN - (X, Y + 1)
                            nextX = currCoords.X;
                            nextY = currCoords.Y + 1;
                            break;
                     case 3:       //RIGHT - (X + 1, Y)
                            nextX = currCoords.X + 1;
                            nextY = currCoords.Y;
                            break;
                    }

                    //If within bounds
                    if (nextX >= 0 && nextX < Global.noOfTilesX && nextY >= 0 && nextY
< Global.noOfTilesY)
                    {
                     Tile nextTile = Global.lstMapRow[nextY][nextX];
                     int nextTileDist = currTileDist + nextTile.getWeight();

                     //If next tile NOT wall
                        if (nextTile.getTileType() != "Wall")
```

```
                    {
                        //If next tile NOT seen
                        if (!tilesSeen.ContainsKey(nextTile.getCoords()))
                        {
                            //If next tile is DESTINATION
                            if (nextTile.getCoords() == endPoint)
                            {
                            //End Search
                              tilesToCheck.Clear();
                                 dir = 3;
                            pathFound = true;
                            }
                            else
                            {
                            //Enqueue next tile, mark as seen
                              tilesToCheck.Add(new Tuple<int,
string>(nextTileDist, nextTile.getCoords()));

                              setOverlayTileStatus(pbxOverlay,
nextTile.getCoords(), Color.Orange);      //Orange - Looking
                              revertTileOverlayColour(ref pbxOverlay,
nextTile.getCoords());
                            }

                           tilesSeen.Add(nextTile.getCoords(), new Tuple<int,
string>(nextTileDist, currTileCoords));
                        }

                        //If tile previously seen, compare existing tentative
distance with new one
                        else
                        {
                         //If tile not yet checked, update tentative distance if
new one is less than current
                            if (tilesToCheck.Contains(nextTile.getCoords()) &&
nextTileDist < tilesSeen[nextTile.getCoords()].Item1)
                           {
                            //MARK nextTile AS UPDATING
                              setOverlayTileStatus(pbxOverlay,
nextTile.getCoords(), Color.Blue);       //Blue - Updating
                              tilesSeen[nextTile.getCoords()] = new Tuple<int,
string>(nextTileDist, currTileCoords);
                              revertTileOverlayColour(ref pbxOverlay,
nextTile.getCoords());
                            }
                        }
                     }
                   }
                }

                  revertTileOverlayColour(ref pbxOverlay, currTileCoords);
              }

            stopWatch.Stop();
```

```
            if (pathFound)
            {
              tbxOutput = string.Format("Path found!");
              outputToTbxOutput(tbxOutput);

              DijkstraOutputPath(pbxOverlay, ref tilesSeen, startPoint, endPoint);
              tbxOutput = string.Format("Shortest path output to screen");
              outputToTbxOutput(tbxOutput);
            }
            else
            {
              //How to solve for multiple nodes? - To SOME destinations
              MessageBox.Show("Could not find path to destination", "No path",
MessageBoxButtons.OK, MessageBoxIcon.Information);

              //Change 'start pathfinding' button to 'reset'
              Button btnStartPathfind =
(Button)this.Controls.Find("btnStartPathfind", false).FirstOrDefault();
              btnStartPathfind.Text = "Reset pathfinding";
            }

          tbxOutput = string.Format("...pathfinding complete in {0} ms",
stopWatch.ElapsedMilliseconds);
          outputToTbxOutput(tbxOutput);
      }

      //Trace shortest path from end to start -Note: There may be multiple shortest
paths, this is one of them
      private void DijkstraOutputPath(PictureBox pbxOverlay, ref Dictionary<string,
Tuple<int, string>> tilesSeen, string startPoint, string endPoint)
      {
            /* Tracing shortest path [Dijkstra]:
            *
            * Push endPoint into shortestPath Stack
            * currCoords = endPoint
            *
            * While currCoords != startPoint
            *      prevCoords = currCoords' previous coordinates from tilesSeen
            *      Push prevCoords into shortestPath
            *      currCoords = prevCoords
            *
            * While shortestPath NOT empty
            *      pathTile = shortestPath.Pop
            *      Mark pathTile as 'Path' and display on overlay
            */

            Stack<string> shortestPath = new Stack<string>();
            string currCoords = endPoint;

          shortestPath.Push(endPoint);

            //Backtrack to startPoint by going through previously connected tiles
from endPoint
```

```
            while (currCoords != startPoint)
            {
              string prevCoords = tilesSeen[currCoords].Item2;
              shortestPath.Push(prevCoords);
              currCoords = prevCoords;
            }

            //Output shortest path
            while (shortestPath.Count > 0)
            {
              string pathCoords = shortestPath.Pop();
              setOverlayTileStatus(pbxOverlay, pathCoords, Color.DarkRed);

              //Retains 'X's on overlay
              if (tilesSeen[pathCoords].Item1 == 0 || pathCoords == endPoint)
              {
                  drawXOnOverlay((Bitmap)pbxOverlay.Image, pathCoords);
                  pbxOverlay.Refresh();
              }
            }
      }



      //BEWARE OF OBTAINING DISTANCES THROUGH VERY LARGE WEIGHTED NODES - OVERFLOW
      private void pathfindAStar(PictureBox pbxOverlay, string startPoint, string
endPoint, string heuristic)
      {
            /* A* Algorithm:
            *
            * Add start point (key) with G-score of 0, and F-score of 0 (val) to
dictionary tilesToCheck
            * Add start point (key) with G-score of 0, F-score of 0 and previous
connected tile of "" (val) dictionary tilesSeen
            *
            * While tilesToCheck NOT empty
            *     currTile = Node with least F-score
            *     Remove this node from dictionary tilesToCheck
            *     Mark currTile as 'Working' and display on overlay
            *
            *     For each nextTile adjacent to currTile
            *           If nextTile within bounds, not marked as seen
            *           If nextTile = endPoint
            *                 Clear tilesToCheck
            *                 pathFound = True
            *           Else
            *                 newGscore = currTile's tentDist + nextTile's weight
            *
            *                 x = Magnitude of x-dist of nextTile from endPoint
            *                 y = Magnitude of y-dist of nextTile from endPoint
            *                 If heuristic = Manhattan
            *                 newHscore = x + y
            *                 Else If heuristic = Euclidean
            *                 newHscore = sqrt(x^2 + y^2)
```

```
                *                   Else if heuristic = Chebyshev
                *                   If x >= y
                *                         newHscore = x
                *                   Else
                *                         newHscore = y

                *                   newFscore = newGscore + newHscore
                *
                *                  tilesToCheck.Add(nextTileCoords, (newGscore,
newFscore));
                *                   Mark nextTile as 'Seen' and display on overlay
                *
                *                   Add nextTile with newGscore, newFscore, and
currTile (as previous connected tile) into tilesSeen
                *
                *             Else If tile previously seen
                *             Mark nextTile as 'Updating' and display on overlay
                *             Compare nextTile's current F-Score with newFscore
                *             Update nextTile's G-score and F-score if newFscore is less
than the current value
                *             Mark nextTile as 'Seen' and display on overlay
                *
                *      Mark currTile as 'Seen' and display on overlay
                *
                * If pathFound
                *      Trace Shortest Path
                * Else
                *      Output "Could not find path to destination"
                */

                PictureBox pbxGrid = (PictureBox)pbxOverlay.Parent;
                Panel pnlScreen = (Panel)pbxGrid.Parent;
                string tbxOutput = "";
            tbxOutput = string.Format("Pathfinding through {0} with A* - {1}...",
pnlScreen.Name, heuristic);
            outputToTbxOutput(tbxOutput);

                Stopwatch stopWatch = new Stopwatch();
            stopWatch.Start();

                //List to hold G-Score, F-Score, Coords - Acts as priority queue and
used to refer to tilesToCheck dictionary
                PriorityListAStar tilesToCheck = new PriorityListAStar();

                //Holds coordinates of tiles seen (key) and (G-Score, F-score, previous
connected tile) (val)
                Dictionary<string, Tuple<int, int, string>> tilesSeen = new
Dictionary<string, Tuple<int, int, string>>();
          bool pathFound = false;

                //Enqueue Start Pos
              tilesToCheck.Add(new Tuple<int, int, string>(0, 0, startPoint));
              tilesSeen.Add(startPoint, new Tuple<int, int, string>(0, 0, ""));
```

```
            while (tilesToCheck.Count() > 0)
            {
              //Dequeue tile
              Tuple<int, int, string> currTile = tilesToCheck.dequeue();
              int currTileG = currTile.Item1;
              int currTileF = currTile.Item2;
            string currTileCoords = currTile.Item3;

              //Index: 0 = x, 1 = y
              Point currCoords = coordsStringToPoint(currTileCoords);

              //Recolour current tile status
              setOverlayTileStatus(pbxOverlay, currTileCoords, Color.LightYellow);
//Yellow - Working tile

              //Look at adjacent tiles - dir = Checking direction
(up/left/down/right) = (0/1/2/3)
              for (int dir = 0; dir <= 3; dir++)
              {
                  int nextX = 0, nextY = 0;

                  switch (dir)
                  {
                   case 0:       //UP - (X, Y - 1)
                          nextX = currCoords.X;
                          nextY = currCoords.Y - 1;
                           break;
                   case 1:       //LEFT - (X - 1, Y)
                          nextX = currCoords.X - 1;
                          nextY = currCoords.Y;
                          break;
                   case 2:       //DOWN - (X, Y + 1)
                          nextX = currCoords.X;
                          nextY = currCoords.Y + 1;
                          break;
                   case 3:       //RIGHT - (X + 1, Y)
                          nextX = currCoords.X + 1;
                          nextY = currCoords.Y;
                          break;
                  }

                  //If within bounds
                  if (nextX >= 0 && nextX < Global.noOfTilesX && nextY >= 0 && nextY
< Global.noOfTilesY)
                  {
                   Tile nextTile = Global.lstMapRow[nextY][nextX];

                   //If next tile NOT wall
                   if (nextTile.getTileType() != "Wall")
                   {
                          //Calculate F-Score
                          int nextTileG = currTileG + nextTile.getWeight();

                          //Heuristic calculation
```

```
                            Point nextTileCoords =
coordsStringToPoint(nextTile.getCoords());
                            Point endPointCoords = coordsStringToPoint(endPoint);
                            int nextTileH = getHScore(nextTileCoords, endPointCoords,
heuristic);

                            int nextTileF = nextTileG + nextTileH;

                            //If next tile NOT seen
                            if (!tilesSeen.ContainsKey(nextTile.getCoords()))
                            {
                                   //If next tile is DESTINATION
                                   if (nextTile.getCoords() == endPoint)
                                   {
                                   //End Search
                                     tilesToCheck.Clear();
                                   dir = 3;
                                   pathFound = true;
                                   }
                                   else
                                   {
                                   //Enqueue next tile, mark as seen
                                  tilesToCheck.Add(new Tuple<int, int,
string>(nextTileG, nextTileF, nextTile.getCoords()));

                                     setOverlayTileStatus(pbxOverlay,
nextTile.getCoords(), Color.Orange);      //Orange - Looking
                                     revertTileOverlayColour(ref pbxOverlay,
nextTile.getCoords());
                                   }

                                tilesSeen.Add(nextTile.getCoords(), new Tuple<int,
int, string>(nextTileG, nextTileF, currTileCoords));
                            }

                            //If tile previously seen, compare existing F-score with
new one
                            else
                            {
                                   //If tile not yet checked, update F-Score if new
one is less than current
                                   if (tilesToCheck.Contains(nextTile.getCoords()) &&
nextTileF < tilesSeen[nextTile.getCoords()].Item2)
                                   {
                                   //MARK nextTile AS UPDATING
                                     setOverlayTileStatus(pbxOverlay,
nextTile.getCoords(), Color.Blue);        //Blue - Updating
                                     tilesSeen[nextTile.getCoords()] = new Tuple<int,
int, string>(nextTileG, nextTileF, currTileCoords);
                                     revertTileOverlayColour(ref pbxOverlay,
nextTile.getCoords());
                                   }
                            }
                     }
```

```
                    }
                }

                revertTileOverlayColour(ref pbxOverlay, currTileCoords);
            }

            stopWatch.Stop();

            if (pathFound)
            {
                tbxOutput = string.Format("Path found!");
                outputToTbxOutput(tbxOutput);

                AStarOutputPath(pbxOverlay, ref tilesSeen, startPoint, endPoint);
                tbxOutput = string.Format("Shortest path output to screen");
                outputToTbxOutput(tbxOutput);
             }
            else
            {
                //How to solve for multiple nodes? - To SOME destinations
                MessageBox.Show("Could not find path to destination", "No path",
MessageBoxButtons.OK, MessageBoxIcon.Information);

                //Change 'start pathfinding' button to 'reset'
                Button btnStartPathfind =
(Button)this.Controls.Find("btnStartPathfind", false).FirstOrDefault();
                btnStartPathfind.Text = "Reset pathfinding";
            }

            tbxOutput = string.Format("...pathfinding complete in {0} ms",
stopWatch.ElapsedMilliseconds);
            outputToTbxOutput(tbxOutput);
        }

        private int getHScore(Point nextTileCoords, Point endPointCoords, string
heuristic)
         {
            int nextTileH = 0;
            int distToEndX = Math.Abs(endPointCoords.X - nextTileCoords.X);
            int distToEndY = Math.Abs(endPointCoords.Y - nextTileCoords.Y);

            if (heuristic == "MANHATTAN")
            {
                nextTileH = distToEndX + distToEndY;
            }
            else if (heuristic == "EUCLIDEAN")
            {
                nextTileH = (int)Math.Round(Math.Sqrt((distToEndX * distToEndX) +
(distToEndY * distToEndY)));
            }
            else if (heuristic == "CHEBYSHEV")
            {
                if (distToEndX >= distToEndY)
                {
```

```
                nextTileH = distToEndX;
            }
            else
            {
                nextTileH = distToEndY;
            }
        }

        return nextTileH;
    }

    private void AStarOutputPath(PictureBox pbxOverlay, ref Dictionary<string,
Tuple<int, int, string>> tilesSeen, string startPoint, string endPoint)
     {
        /* Tracing shortest path [A*]:
        *
        * Push endPoint into shortestPath Stack
        * currCoords = endPoint
        *
        * While currCoords != startPoint
        *      prevCoords = currCoords' previous coordinates from tilesSeen
        *      Push prevCoords into shortestPath
        *      currCoords = prevCoords
        *
        * While shortestPath NOT empty
        *      pathTile = shortestPath.Pop
        *      Mark pathTile as 'Path' and display on overlay
        */

        Stack<string> shortestPath = new Stack<string>();
        string currCoords = endPoint;

       shortestPath.Push(endPoint);

        //Backtrack to startPoint by going through previously connected tiles
from endPoint
        while (currCoords != startPoint)
        {
            string prevCoords = tilesSeen[currCoords].Item3;
            shortestPath.Push(prevCoords);
            currCoords = prevCoords;
        }

        //Output shortest path
        while (shortestPath.Count > 0)
        {
            string pathCoords = shortestPath.Pop();
            setOverlayTileStatus(pbxOverlay, pathCoords, Color.DarkRed);

            //Retains 'X's on overlay
            if (tilesSeen[pathCoords].Item1 == 0 || pathCoords == endPoint)
            {
                drawXOnOverlay((Bitmap)pbxOverlay.Image, pathCoords);
                pbxOverlay.Refresh();
```

```
                }
            }
        }

        private void revertTileOverlayColour(ref PictureBox pbxOverlay, string
tileCoords)
        {
            //Mark nextTile AS SEEN unless path
            if (isNextTilePath(pbxOverlay, tileCoords))
            {
                setOverlayTileStatus(pbxOverlay, tileCoords, Color.DarkRed);
        //DarkRed - Path
            }
            else
            {
                setOverlayTileStatus(pbxOverlay, tileCoords, Color.LightGray);
//Grey - Seen
            }

            //Retains 'X's on overlay
            if (Global.lstPointsToPathfind.Contains(tileCoords))
            {
                drawXOnOverlay((Bitmap)pbxOverlay.Image, tileCoords);
                pbxOverlay.Refresh();
            }
        }

         #endregion

#endregion
```

System Testing

---

1. Display Grid

Objective: The program should display a 2D grid of tiles with a valid 'size' (Number of tiles in a row and column) defined by the user.

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
1A	Testing if multiple pictureboxes, to be displayed in pnlGrid, can each be referred to whilst drawing bitmaps into them, so we can later add multiple screens	Normal	The three test pictureboxes display correctly. The temporary debug method's 'pbxNames' output matches	See 'Image 1.1' and 'Image 1.2'	Pass
1B	Testing best-fit picturebox algorithm with a 5x2 tile map	Normal	The picturebox should correctly display the grid and maximise its size in pnlGrid	See 'Image 1.3'	Outer grid lines are not displayed
1C	Testing best-fit picturebox algorithm with a map that has more tiles than pixels in pnlGrid	Erroneous	The picturebox should not display any image as tileLength = 0	No image displayed	Pass
1D	Testing best-fit picturebox algorithm with a 1x1 tile map (Lower extreme)	Extreme	The picturebox should display a filled rectangle with a black border	See 'Image 1.4'	Outer grid lines are not displayed
1E	Testing best-fit picturebox algorithm with the maximum number of tiles	Extreme	The picturebox should first display the grid	See 'Image '1.5' and 'Image 1.6'	Pass

	which allows both the grid lines and tile to be displayed, and then exceeding this value by 1 (Higher extreme)		correctly, then a black image		
1F	Testing new best-fit picturebox algorithm with a 5x2 tile map	Normal	The picturebox should correctly display the grid and maximise its size in pnlGrid, this time with the outer grid lines	See ‘Image 1.7’	Pass
1G	Testing new best-fit picturebox algorithm with a map that has more tiles than pixels in pnlGrid	Erroneous	The picturebox should not display any image as tileLength = 0	No image displayed	Pass
1H	Testing new best-fit picturebox algorithm with a 1x1 tile map (Lower extreme)	Extreme	The picturebox should display a filled rectangle with a black border	See ‘Image 1.8’	Pass
1I	Testing new best-fit picturebox algorithm with the maximum number of tiles which allows both the grid lines and tile to be displayed, and then exceeding this value by 1 (Higher extreme)	Extreme	The picturebox should first display the grid correctly, then a black image	Approximately the same as ‘Image 1.5’ and ‘Image 1.6’	Pass
1J	Testing grid limit and around it noOfTilesX = 708 noOfTilesY = 565	Extreme	At grid limit, an output should be displayed indicating that I am on it. Beyond the grid limit, the screen should appear black	As expected	Pass

1K	Testing tile limit and around it noOfTilesX = 1417 noOfTilesY = 1130	Extreme Erroneous	At tile limit, an output should be displayed indicating that I am on it. Beyond the tile limit, the screen should not appear at all	As expected	Pass
1L	Testing user input validation with a 5x2 tile map	Normal	The program should accept the user input and display a 5x2 tile grid	As expected	Pass
1M	Testing user input validation with a map that has more tiles and less tiles than pixels in pnlGrid	Erroneous	The program should reject the user input in both cases and display a message box that states the width/height is out of range	As expected	Pass
1N	Testing user input validation with a 1x1 tile map	Extreme	The program should accept the user and display a 1x1 tile grid	As expected	Pass

Images

Image 1.1:

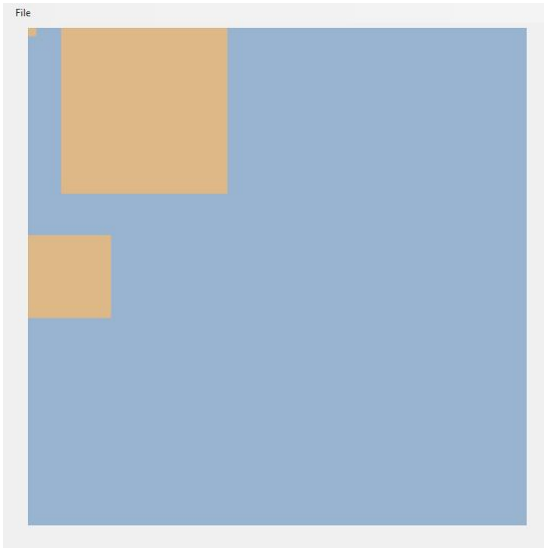


Image 1.2 (Debug output):

(The names from ‘pnlGrid.Controls’ are obtained by looping through each Picturebox in pnlGrid.Controls. If these names are equal to the names stored in lstScreens, then we know that the pictureboxes have all been referred to when drawing bitmaps onto ea.)

Image 1.3:

Image 1.4:

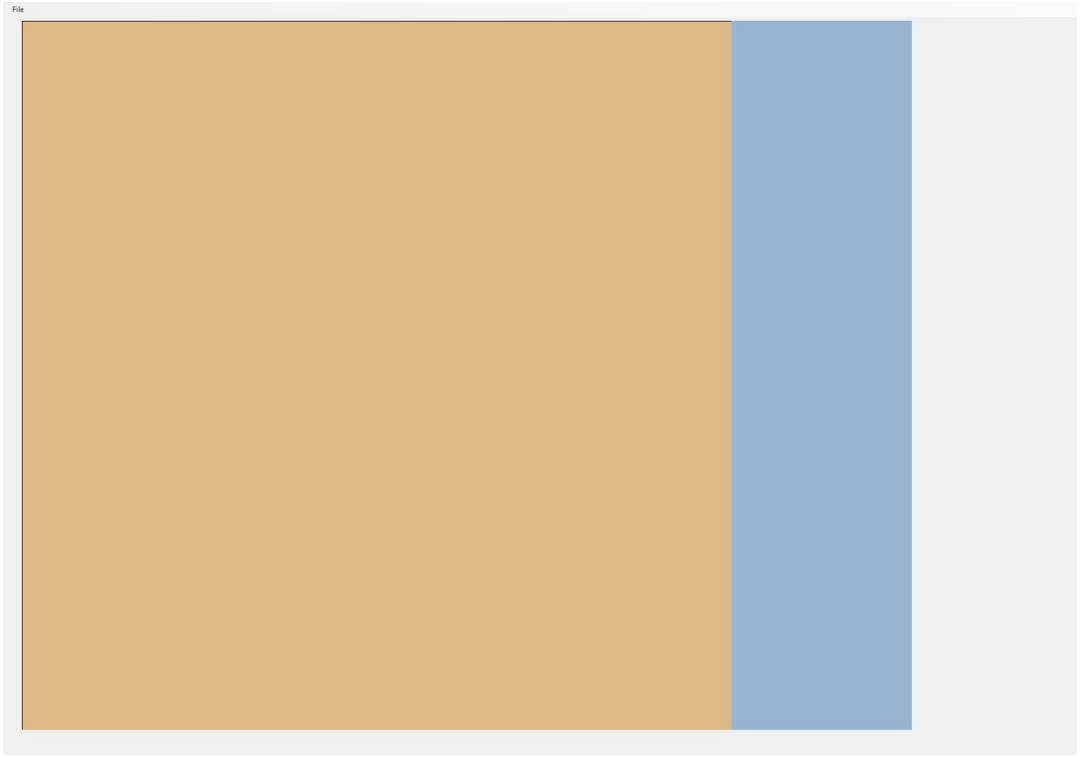


Image 1.5:

Image 1.6:

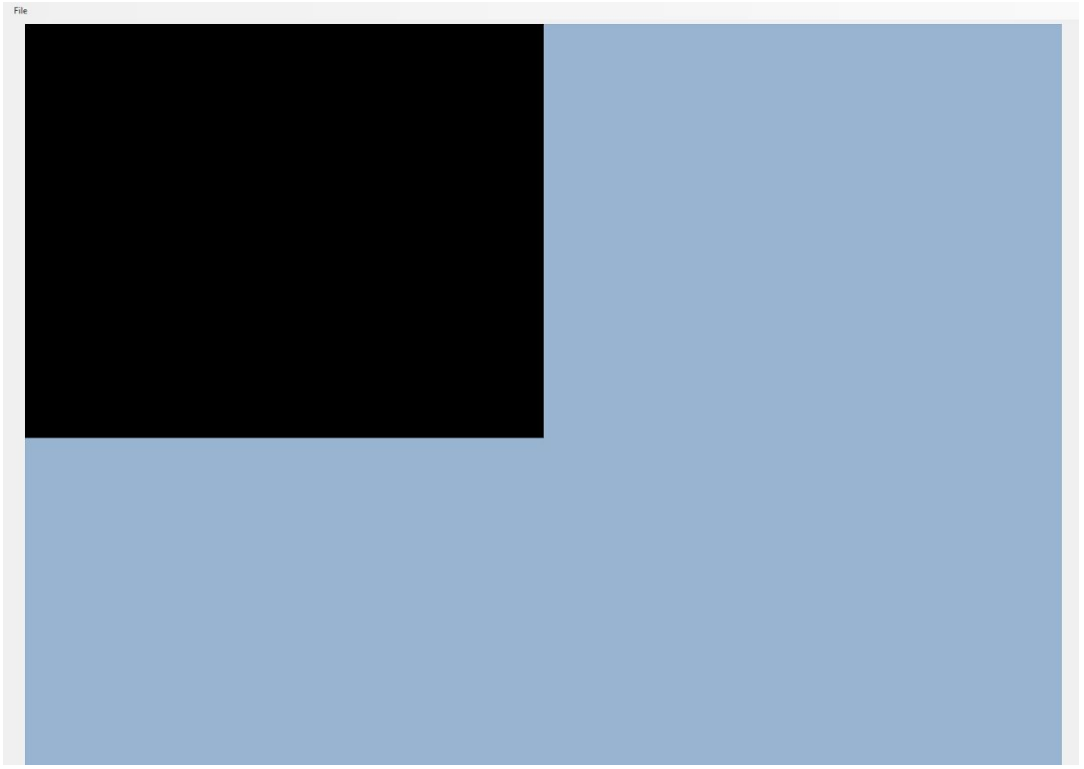
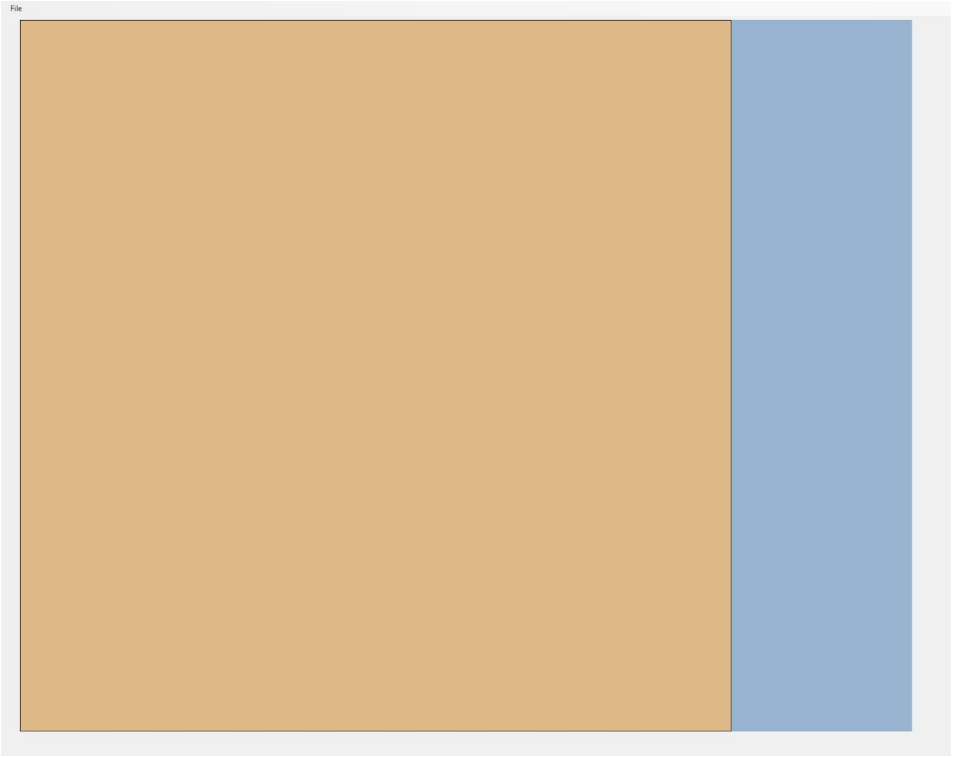


Image 1.7:

Image 1.8:



2. TileTypes

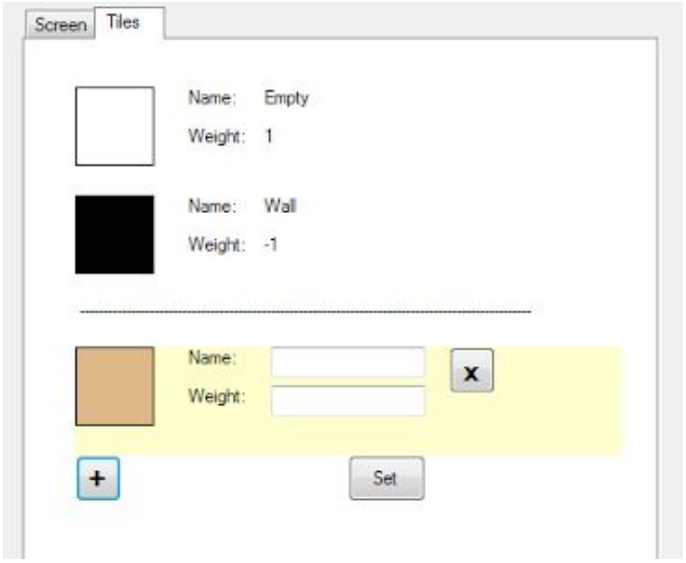
Objective: The user should be able to create their own obstacles with custom name, weight, and colour.

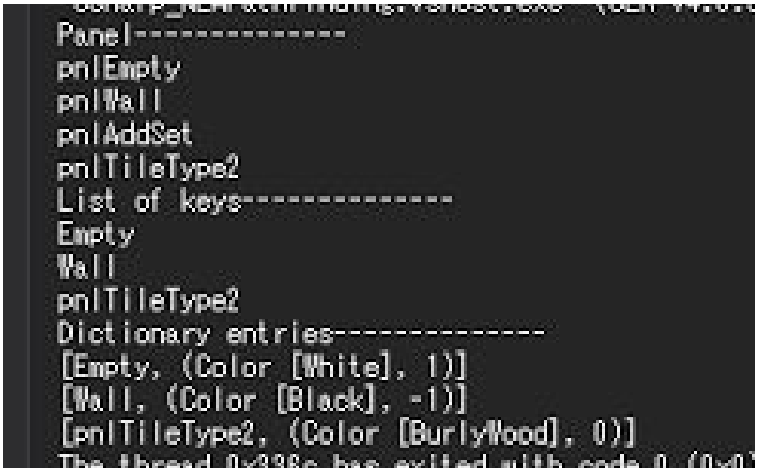
(**Note:** For each of the tests below, their success will be determined by the debug output, coded to output the list of panels, the list of keys for the tile type dictionary, and the value of each key in the dictionary. Some cases (e.g. Colour of panel) will also require visual inspection of the tab page and its panels. These tests will have screenshots provided. However, the change of panel colour due to the tile type properties being changed (but not set) will not be shown.)

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
2A	Adding new tile type panel without setting its values	Normal	See ‘Image 2.1’	As expected	Pass
2B	Deleting last added tile type	Normal	See ‘Image 2.2’	As expected	Pass
2C	Adding two new tile types and deleting the top tile type	Normal	See ‘Image 2.3’	As expected	Pass
2D	Adding two more new tile types and setting the top and bottom tile types	Normal	See ‘Image 2.4’	As expected	Pass
2E	Deleting middle tile type	Normal	See ‘Image 2.5’	As expected	Pass
2F	Adding three more tile types, setting the 2nd last tile type to have the same name as the 1st with a valid weight, and the last tile type to have a valid name, but an invalid weight	Normal Erroneous	See ‘Image 2.6’	As expected	Pass
2G	Deleting 2nd tile type	Normal	See ‘Image 2.7’	As expected	Pass

Images

Image 2.1:



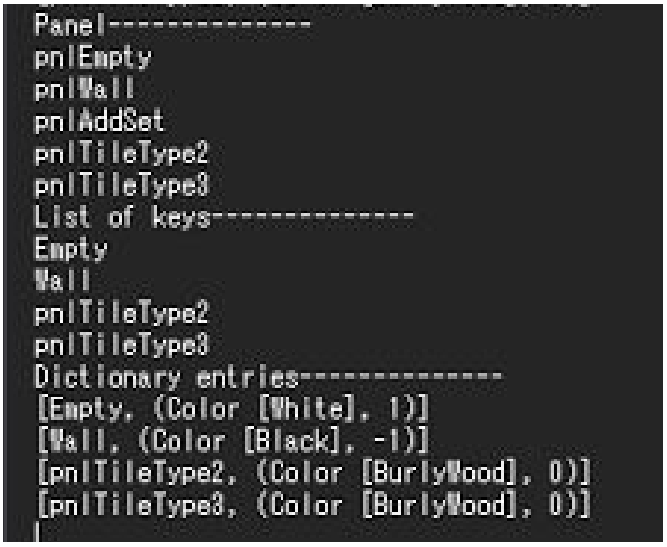


(**Note:** ‘pnlAddSet’ is a panel containing the buttons required to add and set new tile types, inidicated by the ‘+’ and ‘Set’ button texts. The ‘Panel’ list comes from a for loop through all panels in ‘tabTiles’. The order at which the panel names are outputted is based on their time of creation, which is why ‘pnlAddSet’ is above the custom tile type ‘pnlTileType2’. However this should not matter because you can only add tile types from the bottom, so the custom tile types will still be outputted in the correct order.

Additionally, note that the debug images’ panel names are different from the final program - ‘pnlEmpty’ should be ‘Empty’ and ‘pnlWall’ should be ‘Wall’.)

Image 2.2:

Image 2.3:
Before top tile type deleted:



After deletion:

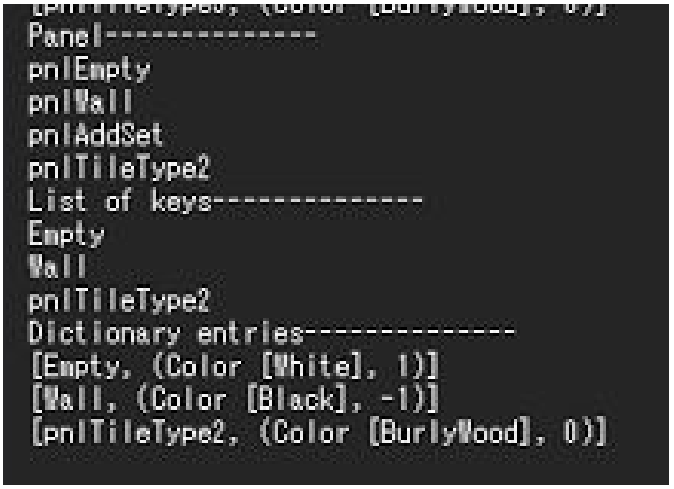
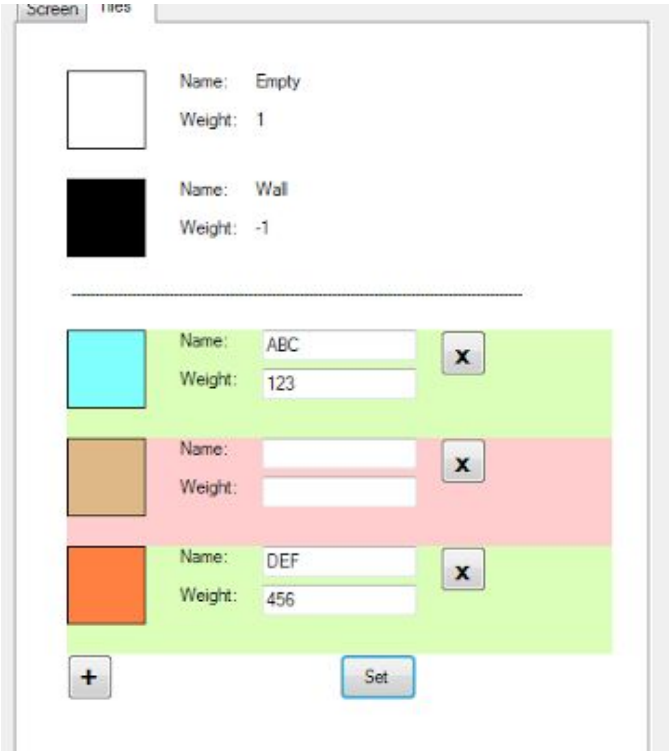
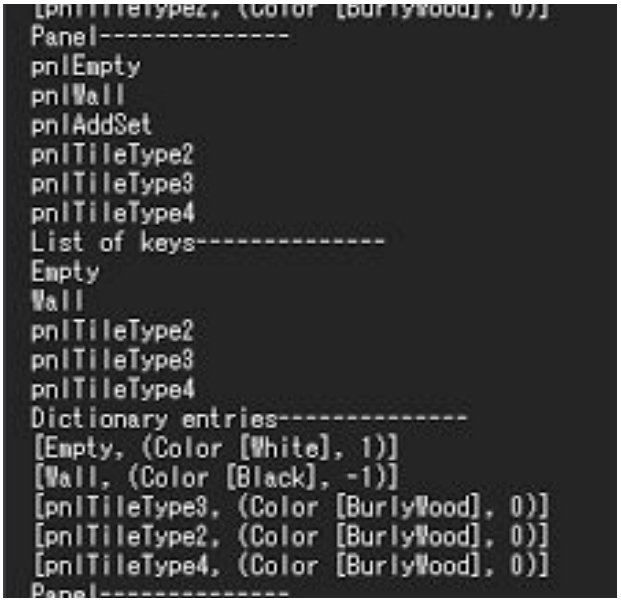


Image 2.4:



Before setting:



After setting:

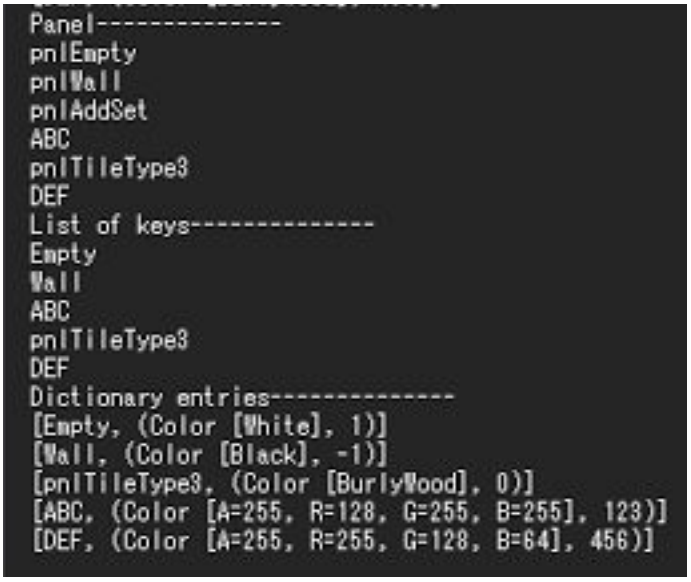


Image 2.5:

After deletion:

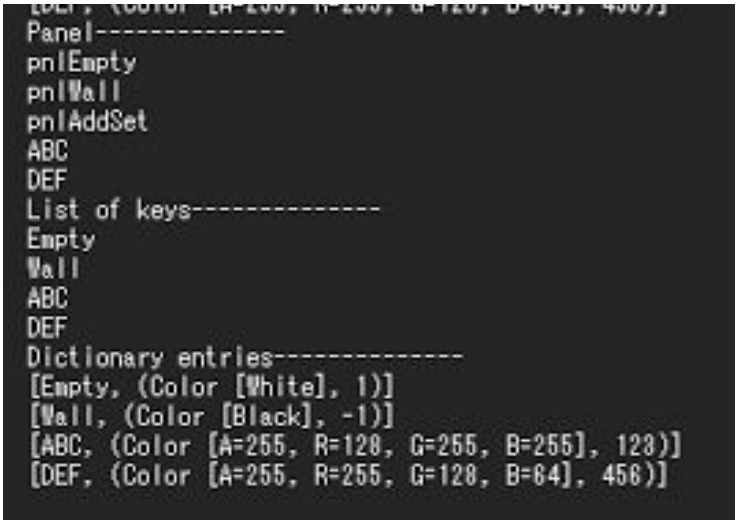
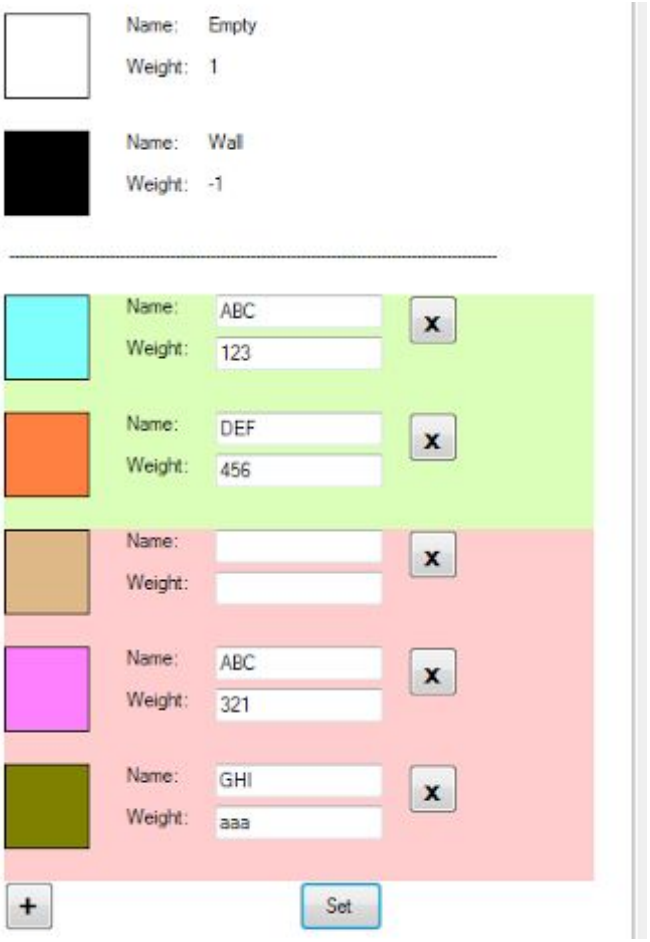


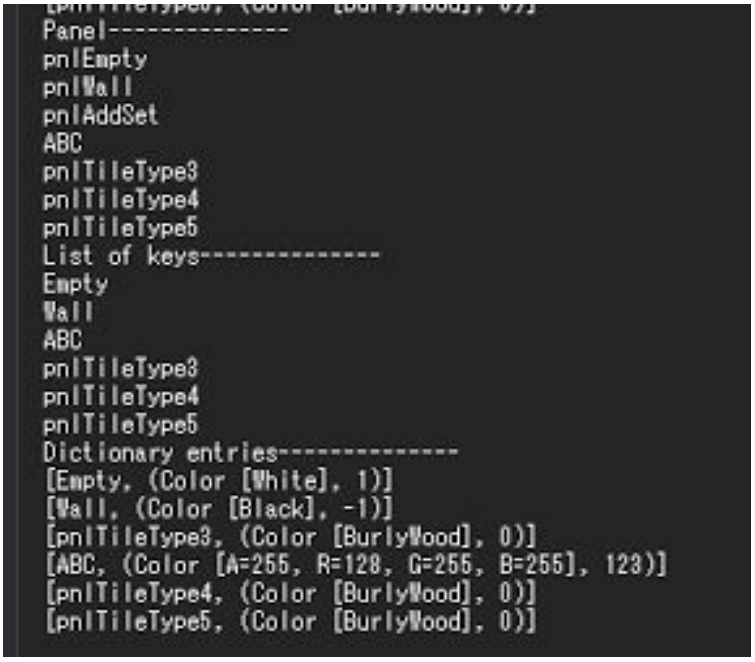
Image 2.6:



Before setting:

After setting (Same as before setting)

Image 2.7:
After deletion:



3. Mouse-Hover Tile Highlight

Objective: When the mouse hovers above a tile, that tile should be highlighted

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
3A	Tile highlight when hovering above tile on screen. Highlight is drawn directly to screen	Normal Extreme Erroneous	On mouse move inside screen, the 'highlight picturebox' should snap to the tile the cursor is hovering above. If the cursor is outside the screen, the picturebox should not display at all	See 'Image 3.1'	Moving mouse too fast creates multiple highlights on screen On outside of screen, highlights do not go away until screen is refreshed This solution is too unreliable. New solution, described in test 3B, attemped
3B	Tile highlight when hovering above tile on screen. Highlight is drawn to transparent bitmap, and added as a child control of the screen	Normal Extreme Erroneous	On mouse move inside screen, the 'highlight picturebox' should snap to the tile the cursor is hovering above. If the cursor is outside the screen, the picturebox should not display at all	Works for a while, then 'Parameter is not valid' crash	I suspect that new bitmaps are being created so quickly that the garbage collector does not have time to free up memory from unused bitmaps, causing a memory leak. New solution, described in test 3C, attempted

3C	‘Overlay’ picturebox is placed on top of screen, completely covering it. When the cursor is hovering above a tile, a tile highlight should be drawn on the overlay, located directly above the tile in the screen. Testing by drawing a red highlight with transparent background on the overlay	Normal Extreme Erroneous	On mouse move inside overlay, the highlight should snap to the tile the cursor is hovering above. If the cursor is outside the overlay, the highlight should not display at all	Grid lines remain on overlay after moving to next tile	See test 3D for reason
3D	Testing overlay by drawing a transparent black rectangle (colour black with alpha = 1) on the overlay	Normal	On mouse move inside overlay, the tile which the mouse is hovering above should be highlighted in a faint grey colour	See ‘Image 3.2’	The colours are being blended rather than replaced. Problem solved by changing graphics compositing mode from ‘SourceOver’ to ‘SourceCopy’
3E	Testing mouse-hover tile highlight with overlay on ‘outer grid lines’ (bottom-most and right-most grid lines on the screen)	Extreme	On outer grid line, the highlight should be on top of the tile next to the grid line	See ‘Image 3.3’	Corrected for this by subtracting the highlighted tile value by 1 if the cursor is on a outer grid line
3F	Testing mouse-hover tile highlight with overlay on full map	Normal Extreme Erroneous	On mouse move inside overlay, the highlight should snap to the tile the cursor is hovering above. If the cursor is outside the overlay, the highlight should not display at all	As expected	Pass

3G	Obtaining information of tile clicked Testing with 11th column, 6th row, black tile	Normal	When the left mouse button is clicked, the highlighted tile's coordinates should be outputted in a messagebox	See 'Image 3.4'	Pass
3H	Obtaining information of corner tiles when clicked Showing top right corner (20th column, 1st row), white tile	Extreme	When the left mouse button is clicked, the highlighted tile's coordinates should be outputted in a messagebox	See 'Image 3.5'	Pass
3I	Holding mouse down whilst moving outside of overlay	Erroneous	No crashes should happen	Crash - Out of range exception thrown	This is because the program is attempting to declare a tile that is out of bounds (in order to obtain its information) Tile coordinate values obtained through cursor position is restricted to inside the overlay only
3J	Holding mouse down whilst moving outside of overlay	Erroneous	No crashes should happen	As expected	Pass

Images

Image 3.1:

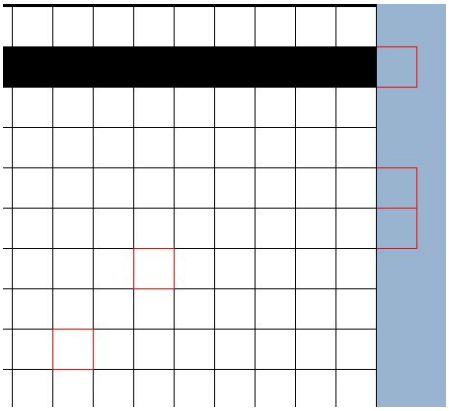


Image 3.2:

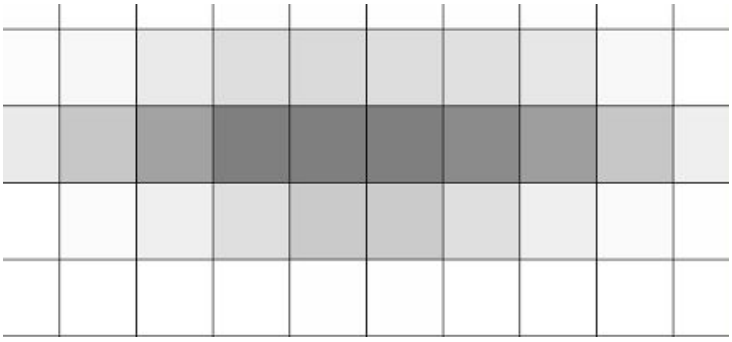


Image 3.3:

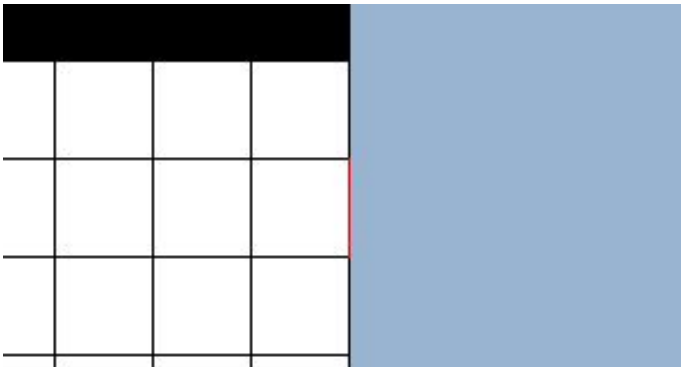
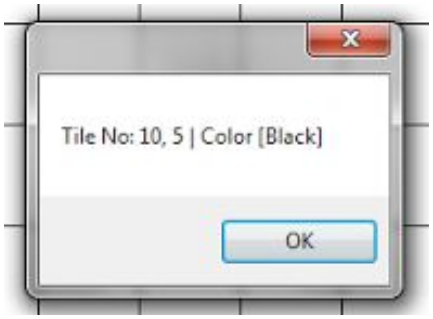
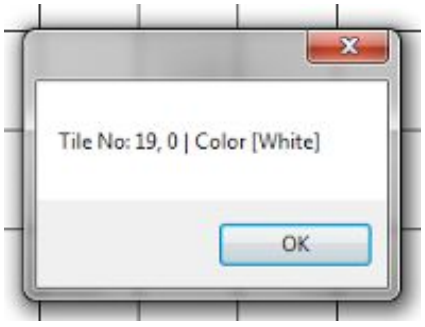


Image 3.4:



(**Note:** (x,y) both start from 0)

Image 3.5:



4. Resizing Map

Objective: The user should be able to resize the map to a chosen number of tiles in X and Y.

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
4A	Testing map output to text file, for next tests	Normal	Coordinates and tile type of each tile on the map is outputted, separately, to a text file	See ‘Image 4.1’ for map image See ‘Image 4.2’ for text file output	Pass
4B	Map shrunk to 5x5	Normal	A subset of the original 10x10 map should be displayed	See ‘Image 4.3’	Pass
4C	Map enlarged back to 10x10	Normal	The previously coloured tiles, deleted when the map was shrunk, should now be empty	See ‘Image 4.4’ and ‘Image 4.5’	Pass

Images

Image 4.1:

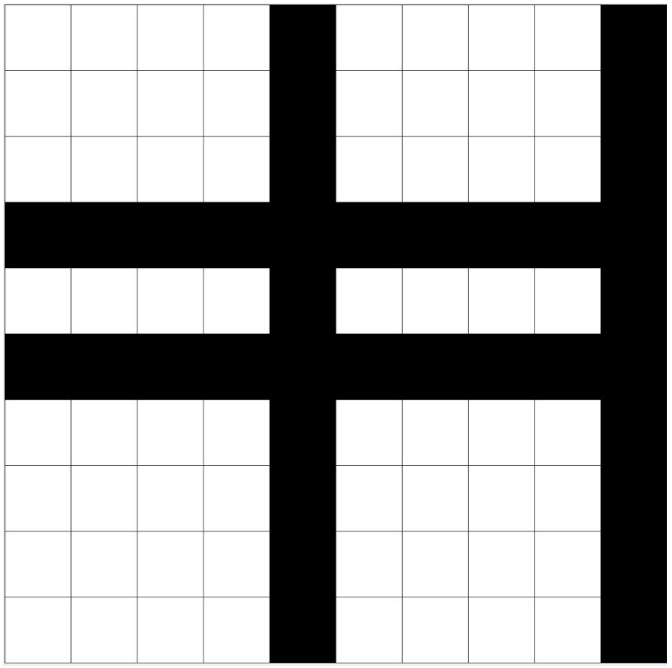


Image 4.2:

```
0,0 1,0 2,0 3,0 4,0 5,0 6,0 7,0 8,0 9,0
0,1 1,1 2,1 3,1 4,1 5,1 6,1 7,1 8,1 9,1
0,2 1,2 2,2 3,2 4,2 5,2 6,2 7,2 8,2 9,2
0,3 1,3 2,3 3,3 4,3 5,3 6,3 7,3 8,3 9,3
0,4 1,4 2,4 3,4 4,4 5,4 6,4 7,4 8,4 9,4
0,5 1,5 2,5 3,5 4,5 5,5 6,5 7,5 8,5 9,5
0,6 1,6 2,6 3,6 4,6 5,6 6,6 7,6 8,6 9,6
0,7 1,7 2,7 3,7 4,7 5,7 6,7 7,7 8,7 9,7
0,8 1,8 2,8 3,8 4,8 5,8 6,8 7,8 8,8 9,8
0,9 1,9 2,9 3,9 4,9 5,9 6,9 7,9 8,9 9,9

Empty Empty Empty Empty Wall Empty Empty Empty Empty Wall
Empty Empty Empty Empty Wall Empty Empty Empty Empty Wall
Empty Empty Empty Empty Wall Empty Empty Empty Empty Wall
Wall Wall Wall Wall Wall Wall Wall Wall Wall Wall
Empty Empty Empty Empty Wall Empty Empty Empty Empty Wall
Wall Wall Wall Wall Wall Wall Wall Wall Wall Wall
Empty Empty Empty Empty Wall Empty Empty Empty Empty Wall
Empty Empty Empty Empty Wall Empty Empty Empty Empty Wall
Empty Empty Empty Empty Wall Empty Empty Empty Empty Wall
Empty Empty Empty Empty Wall Empty Empty Empty Empty Wall
```

(**Note:** Only doing 10x10 map at max, otherwise it will be difficult to line up and compare the coordinates)

Image 4.3:

```
0,0 1,0 2,0 3,0 4,0
0,1 1,1 2,1 3,1 4,1
0,2 1,2 2,2 3,2 4,2
0,3 1,3 2,3 3,3 4,3
0,4 1,4 2,4 3,4 4,4

Empty Empty Empty Empty Wall
Empty Empty Empty Empty Wall
Empty Empty Empty Empty Wall
Wall Wall Wall Wall Wall
Empty Empty Empty Empty Wall
```

Image 4.4:

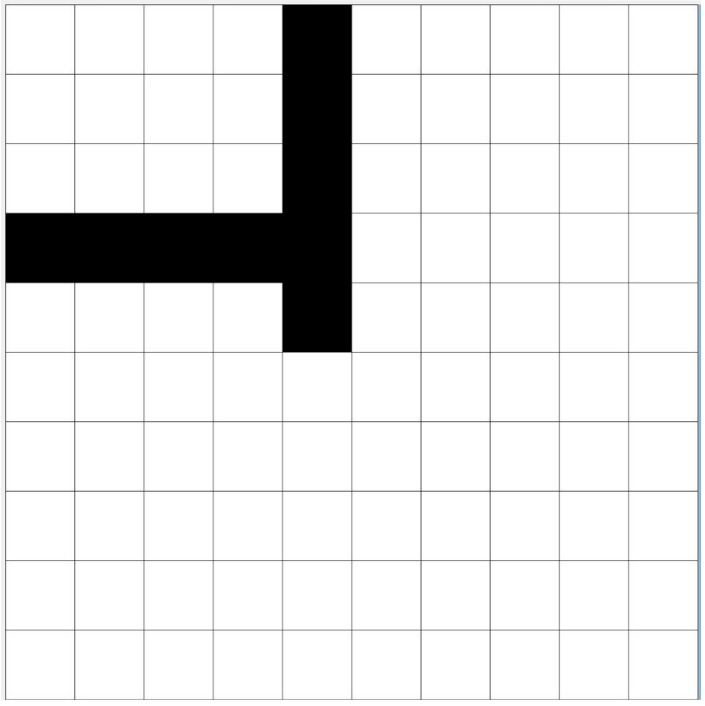


Image 4.5:

```
0,0 1,0 2,0 3,0 4,0 5,0 6,0 7,0 8,0 9,0
0,1 1,1 2,1 3,1 4,1 5,1 6,1 7,1 8,1 9,1
0,2 1,2 2,2 3,2 4,2 5,2 6,2 7,2 8,2 9,2
0,3 1,3 2,3 3,3 4,3 5,3 6,3 7,3 8,3 9,3
0,4 1,4 2,4 3,4 4,4 5,4 6,4 7,4 8,4 9,4
0,5 1,5 2,5 3,5 4,5 5,5 6,5 7,5 8,5 9,5
0,6 1,6 2,6 3,6 4,6 5,6 6,6 7,6 8,6 9,6
0,7 1,7 2,7 3,7 4,7 5,7 6,7 7,7 8,7 9,7
0,8 1,8 2,8 3,8 4,8 5,8 6,8 7,8 8,8 9,8
0,9 1,9 2,9 3,9 4,9 5,9 6,9 7,9 8,9 9,9

Empty Empty Empty Empty Wall Empty Empty Empty Empty Empty
Empty Empty Empty Empty Wall Empty Empty Empty Empty Empty
Empty Empty Empty Empty Wall Empty Empty Empty Empty Empty
Wall Wall Wall Wall Wall Empty Empty Empty Empty Empty
Empty Empty Empty Empty Wall Empty Empty Empty Empty Empty
Empty Empty Empty Empty Empty Empty Empty Empty Empty Empty
Empty Empty Empty Empty Empty Empty Empty Empty Empty Empty
Empty Empty Empty Empty Empty Empty Empty Empty Empty Empty
Empty Empty Empty Empty Empty Empty Empty Empty Empty Empty
Empty Empty Empty Empty Empty Empty Empty Empty Empty Empty
```

5. Edit Map Tiles

Objective: The user should be able to edit the tiles of the map using the grid.

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
5A	Deleting a custom tileType while a tile of that tileType exists on the grid	Normal	All tiles of the deleted tileType should be deleted off the map and removed from display	“Given key was not present in dictionary” error when attempting to redraw map	Created a new dictionary that will have the same key as the current dictionary, but stores a HashSet as its value. This HashSet holds the coordinates of tiles for each tileType
5B	Added the following custom tileTypes and then pressed ‘Set’: *[Name, Weight, Colour]* [, , Red] [ABC, 123, Orange] [DEF, , Yellow] [GHI, 456, Green] [, 789, Blue] [JKL, 42, Pink] [ABC, 1337, Purple]	Normal Erroneous	When the button ‘Set’ is clicked, only the valid tileTypes should be set and its tileType panels should turn green. The rest of the panels should turn red	“Given key was not present in dictionary” error See ‘Image 5.1’ for dictionary state before crash	Error was due to looking up tileType ABC’s colour, even though a dicTileTypeTiles entry for that tileType has not yet been created. Solved by restricting colour update to existing tileTypes only
5C	Same as 5B	Normal Erroneous	Same as 5B	See ‘Image 5.2’ for the tileTypes tab page and ‘Image 5.3’ for the dictionary state after setting	Pass
5D	The invalid tileTypes are deleted	Normal	The dictionary entries of each tileType should be removed	See ‘Image 5.4’ for the dictionary state after deletion	Pass

5E	The map is shrunk to 5x5 and some tiles are coloured using remaining tileTypes	Normal	The coordinates of each coloured tile should be added to the corresponding tileType key in the dictionary	See ‘Image 5.5’ for the coloured map and ‘Image 5.6’ for the coordinates stored in the dictionary for each custom tileType	Pass
5F	The colours of each tileType is changed	Normal	For each tile that has its tileType colour changed, the program should redraw the tiles with the updated colour	As expected	Pass
5G	Each custom tileType is deleted	Normal	As each tileType is deleted, its dictionary entries should be removed and tiles of the deleted tileType in the map should be replaced with empty tiles	As expected	Pass

5H	An empty tileType is added and then deleted	Normal	The tileType should be deleted and the program should not crash	The program crashes as it attempted delete the dictionary entry that holds the tile coordinates of that tileType, even though it hasn't been created yet	Added missing if-statement into the code
5I	Same as 5H	Normal	Same as 5H	As expected	Pass

Images

Image 5.1:

```
List of keys--------------
Empty
Wall
pnlTileType2
pnlTileType3
pnlTileType4
pnlTileType5
pnlTileType6
pnlTileType7
pnlTileType8
Dictionary Info entries--------------
[Empty, (Color [White], 1)]
[Wall, (Color [Black], -1)]
[pnlTileType2, (Color [BurlyWood], 0)]
[pnlTileType3, (Color [BurlyWood], 0)]
[pnlTileType4, (Color [BurlyWood], 0)]
[pnlTileType5, (Color [BurlyWood], 0)]
[pnlTileType6, (Color [BurlyWood], 0)]
[pnlTileType7, (Color [BurlyWood], 0)]
[pnlTileType8, (Color [BurlyWood], 0)]
Dictionary Tiles entries--------------
[Empty, System.Collections.Generic.HashSet`1[System.String]]
[Wall, System.Collections.Generic.HashSet`1[System.String]]
```

Image 5.2:

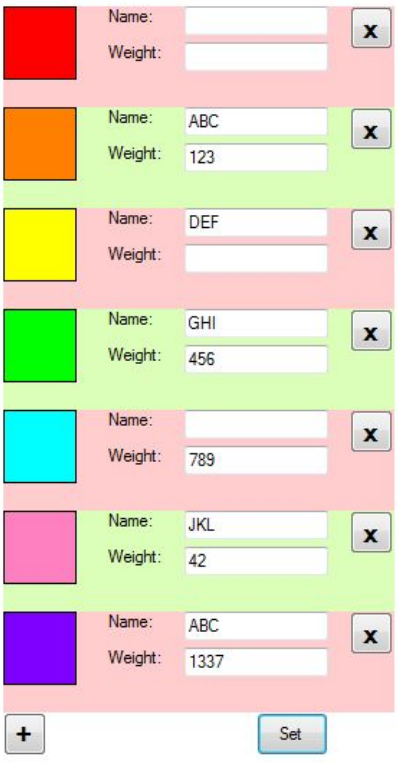


Image 5.3:

```
List of keys--------------
Empty
Wall
pnlTileType2
ABC
pnlTileType4
GHI
pnlTileType6
JKL
pnlTileType8
Dictionary Info entries--------------
[Empty, (Color [White], 1)]
[Wall, (Color [Black], -1)]
[pnlTileType2, (Color [BurlyWood], 0)]
[ABC, (Color [A=255, R=255, G=128, B=0], 123)]
[pnlTileType4, (Color [BurlyWood], 0)]
[GHI, (Color [Lime], 456)]
[pnlTileType6, (Color [BurlyWood], 0)]
[JKL, (Color [A=255, R=255, G=128, B=192], 42)]
[pnlTileType8, (Color [BurlyWood], 0)]
Dictionary Tiles entries--------------
[Empty, System.Collections.Generic.HashSet`1[System.String]]
[Wall, System.Collections.Generic.HashSet`1[System.String]]
[ABC, System.Collections.Generic.HashSet`1[System.String]]
[GHI, System.Collections.Generic.HashSet`1[System.String]]
[JKL, System.Collections.Generic.HashSet`1[System.String]]
```

Image 5.4:

```
List of keys--------------
Empty
Wall
ABC
GHI
JKL
Dictionary Info entries--------------
[Empty, (Color [White], 1)]
[Wall, (Color [Black], -1)]
[ABC, (Color [A=255, R=255, G=128, B=0], 123)]
[GHI, (Color [Lime], 456)]
[JKL, (Color [A=255, R=255, G=128, B=192], 42)]
Dictionary Tiles entries--------------
[Empty, System.Collections.Generic.HashSet`1[System.String]]
[Wall, System.Collections.Generic.HashSet`1[System.String]]
[ABC, System.Collections.Generic.HashSet`1[System.String]]
[GHI, System.Collections.Generic.HashSet`1[System.String]]
[JKL, System.Collections.Generic.HashSet`1[System.String]]
```

Image 5.5:

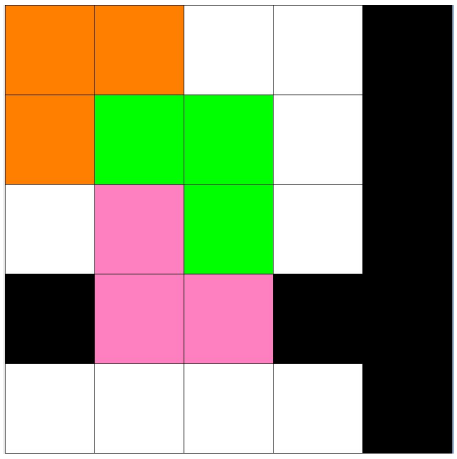


Image 5.6:

```
[ABC, System.Collections.Generic.HashSet`1[System.String]]
0,0
0,1
1,0
[GHI, System.Collections.Generic.HashSet`1[System.String]]
1,1
2,2
2,1
[JKL, System.Collections.Generic.HashSet`1[System.String]]
1,2
1,3
2,3
```

6. Clear Map and Add Flags

Objective: The user should be able to clear the map and add a start and end flag to the grid. The flags will indicate between which points the shortest path will be calculated.

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
6A	Temporarily making the default noOfTilesXY to be 4x4, valid custom tileTypes are created and used to draw some custom tiles on the map. Additional empty panels are then added and the map is resized to 100x100. After this, the option to create a new project is clicked	Normal	All custom tileType panels should be deleted. The list of dictionary keys and the dictionaries should be cleared of all tileTypes except the default ones. dicTileTypeTiles should have all tiles in key “Empty”	As expected See ‘Image 6.1’ for final debug output	Pass
6B	Custom tile types are added and used to draw some custom tiles on the map. The map is then cleared	Normal	Only the map should be cleared. Any custom tile types should remain	As expected	Pass
6C	Hovering over set’ path point’ (Tiles to path find between)	Normal	The highlighted path point should not be affected when hovering across it	When adjacent tiles are highlighted, one of the path point tile edges is deleted. See ‘Image 6.2’	This is because adjacent tiles share a common grid line New indicator for selected path point created and tested in 6D

6D	Setting path points	Normal	When a tile is clicked, and ‘path point mode’ is selected, a ‘X’ should appear on the clicked tile	As expected See ‘Image 6.3’	Pass This new indicator should not be affected when hovering as none of the cross sits on the grid line
6E	Resizing map with path points on map	Normal	The path points should remain on the tile it was placed at on resize	Artifacts appearing on screen See ‘Image 6.4’ and ‘Image 6.5’	Attempted to find problem in test 6F
6F	Finding cause of artifacts on screen when resizing map with path points on map	Normal	I added “debug writelines” to the program to output properties of the program at a specific instance of runtime	Found that the grid and overlay size did not match up after resize See ‘Image 6.6’	Solved by deleting existing overlay in grid before adding new one

Images

Image 6.1:

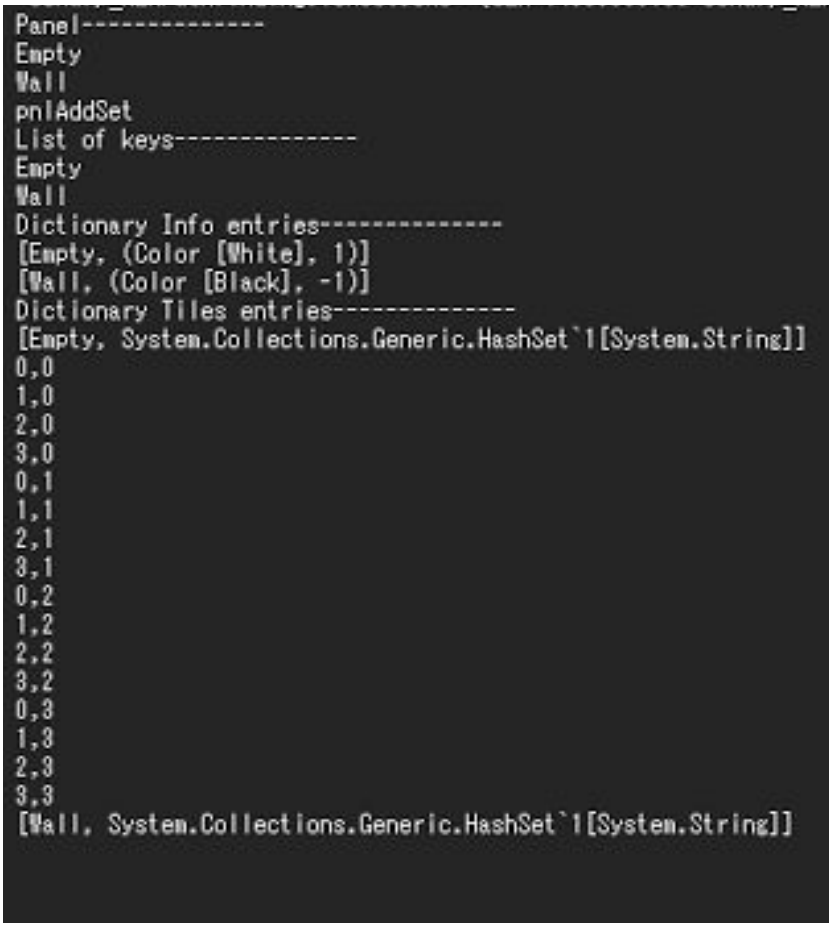


Image 6.2:

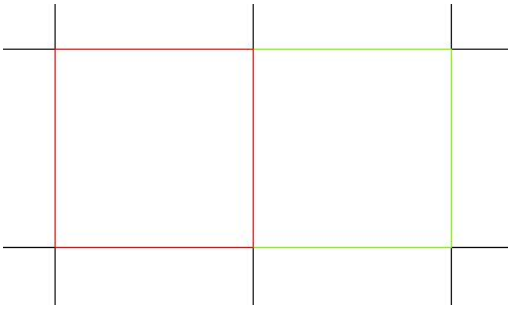


Image 6.3:

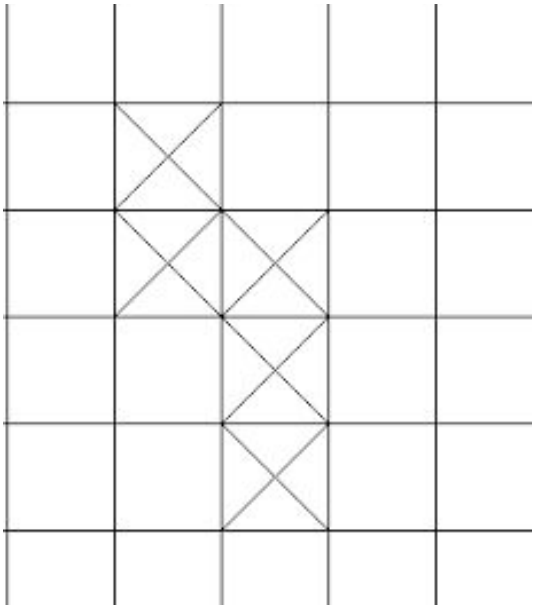


Image 6.4:

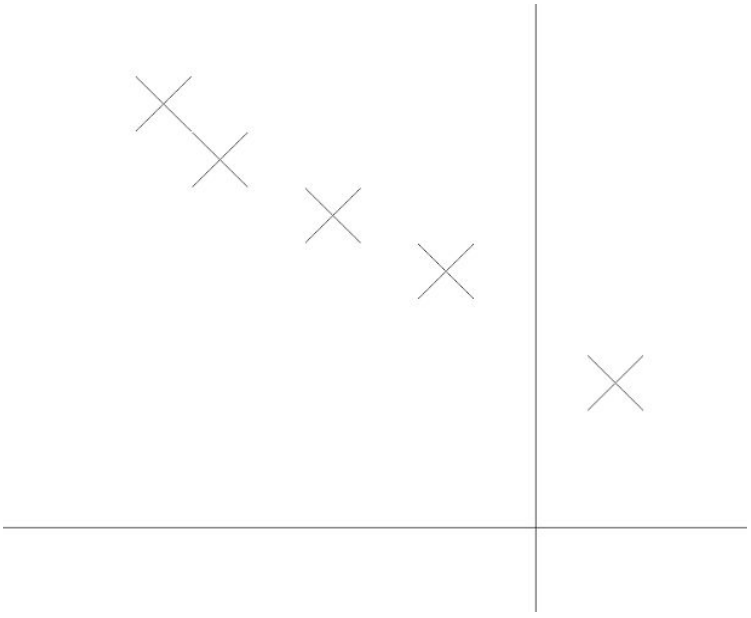


Image 6.5:

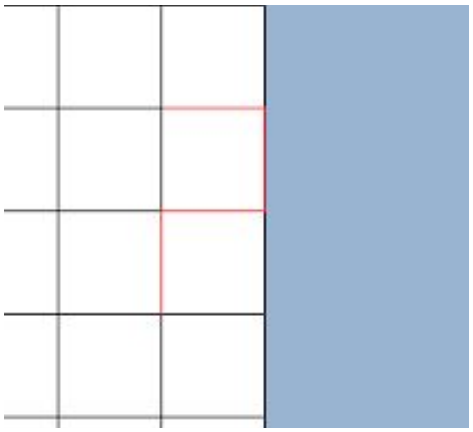


Image 6.6:

7. BFS Algorithm and Boundary Fill

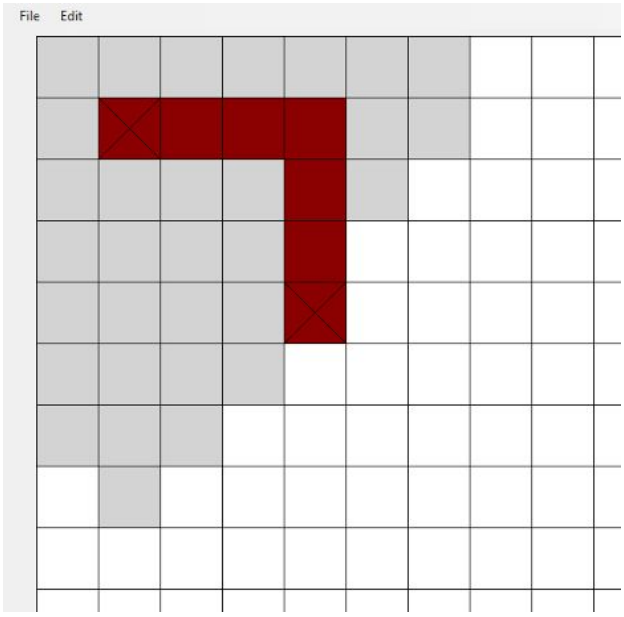
Objective: The user should be able to pathfind between the set path points using Breadth-First Search (BFS) Algorithm. The user should also be able to fill tiles within a particular bounded area to a chosen tileType using the "Boundary Fill'" variation of the BFS Algorithm.

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
7A	Perform BFS and draw shortest path between start and end point	Normal	Zig-Zag from start point towards end point drawn	See 'Image 7.1'	Pass
7B	Added an obstacle in the shortest path drawn to see how it affects the new shortest path drawn	Normal	There should be a shortest path that goes around the obstacle	As expected See 'Image 7.2'	Pass
7C	A large obstacle is placed in between the start and end point and the BFS algorithm is run	Normal	There should be a shortest path that goes around the obstacle	See 'Image 7.3'	There is an infinite loop where items constantly get added to the queue of tiles to check and the program does not end as the queue will not be empty. This problem is due to adding tiles that have been previously visited

7D	Same as 7C	Normal	There should be a shortest path that goes around the obstacle	See ‘Image 7.4’	Pass Problem solved by adding a particular layer number to each tile, determined by its distance from the start tile. All tiles searched must be connected to another tile from a previous layer. This fact is used to find the layer number of all tiles, starting from the start tile with a layer number of 0
7E	Testing what happens if a path is not found	Erroneous	A message box should appear saying that no path could be found to the destination	As expected See ‘Image 7.5’	Pass
7F	Testing what would happen if the algorithm had to find a path through custom tileTypes	Normal	The algorithm should treat custom tileTypes as impassable (walls)	As expected See ‘Image 7.6’	Pass
7G	Using boundary fill algorithm to fill shapes with colour (replace all tiles in a certain region with a selected tileType)	Normal	The shapes should be filled with a selected tileType and its colour on the grid should be updated	As expected See ‘Image 7.7’ and ‘Image 7.8’	Pass
7H	Increased map size after filling shape	Normal	The filled shape should still be retained after increasing the map size	As expected See ‘Image 7.9’	Pass

Images

Image 7.1:



Although the most natural solution here is a zig-zag towards the destination, this is still the shortest path as movement is restricted to up/down/left/right, thus diagonal zig-zag motion will have same weight (pass through the same number of tiles) as the path shown above.

Image 7.2:

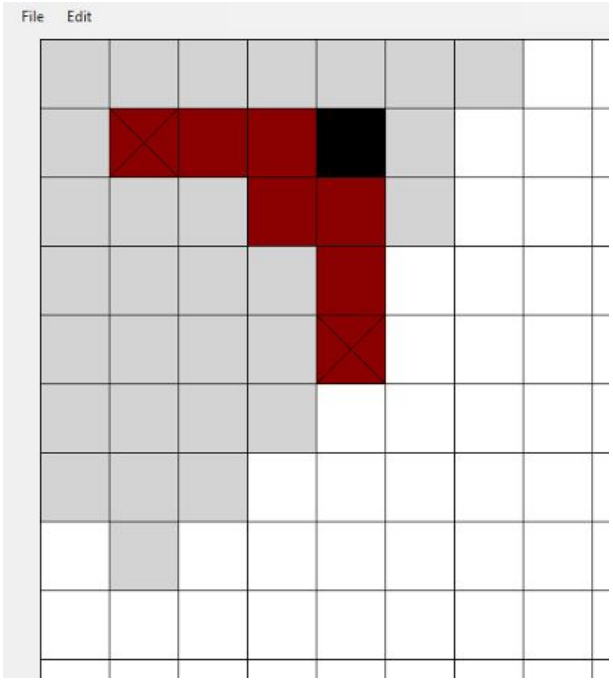


Image 7.3:

Image 7.4:

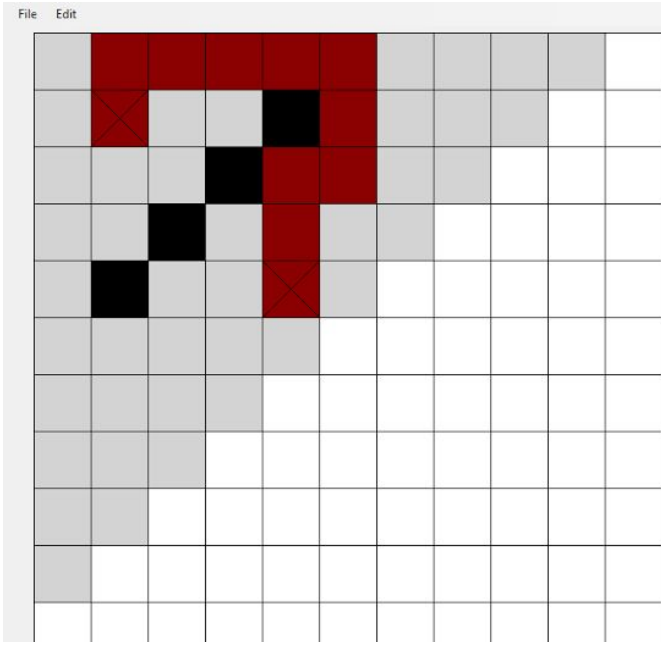


Image 7.5:

Image 7.6:

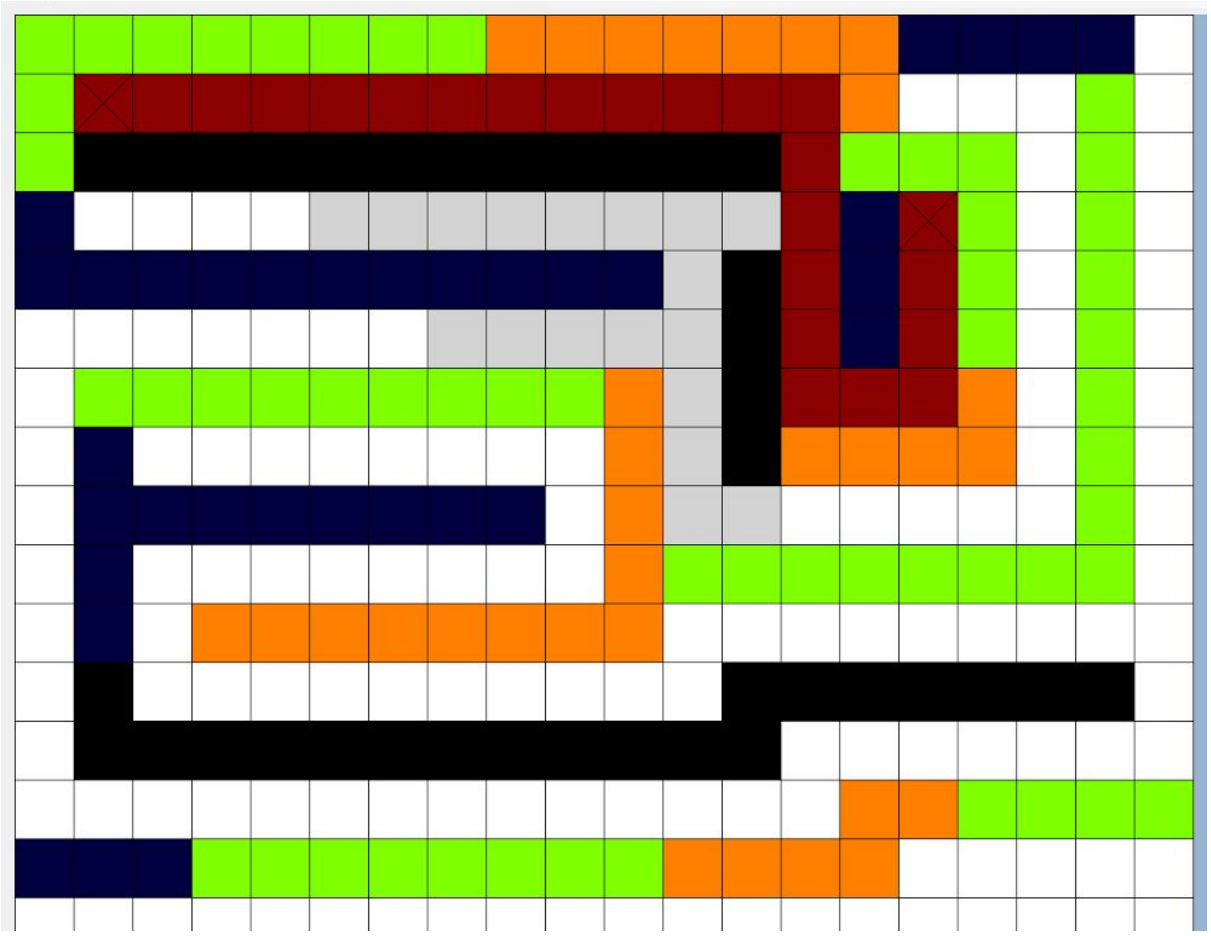


Image 7.7:

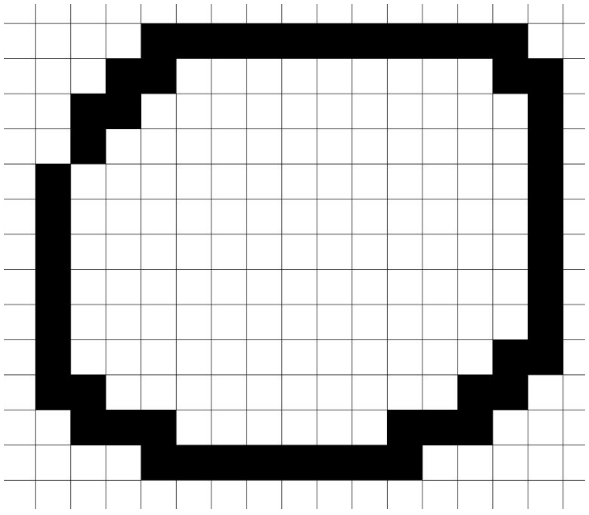


Image 7.8:

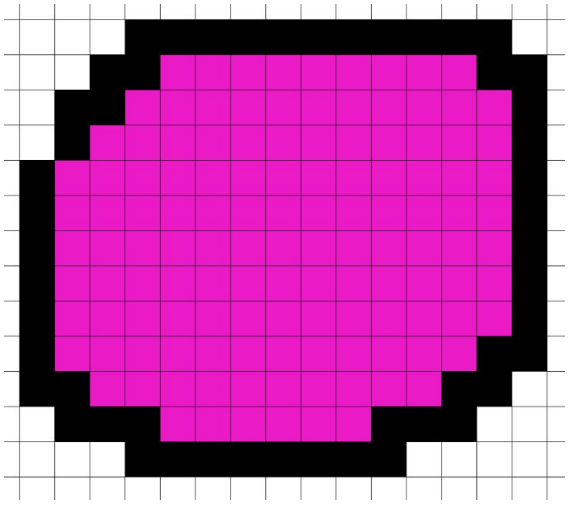
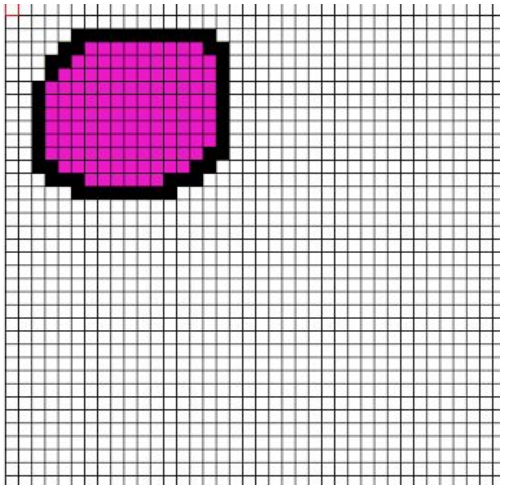


Image 7.9:



8. Multi-Screen Pathfinding Comparison

Objective: The user should be able to compare multiple pathfinding algorithms of their choice by displaying copies of the map, each with a different pathfinding algorithm running. By pressing 'play', all chosen pathfinding algorithms should start simultaneously.

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
8A	Testing deletion of additional screen panels	Normal	Panel deletes and any panels below the deleted panel has its name and locations shifted up by one panel	'Image 8.1' exception thrown when attempting to change panel name. Debug output of list of screen names shown in 'Image 8.2'	The first screen had a name of 'pbxGrid' rather than 'Screen0', thus causing the screen deletion to be off by one screen
8B	Testing deletion of additional screen panels	Normal	Same as Test 8A	As expected	Pass
8C	Testing the improved best-fit picturebox algorithm when adding screens	Normal	From the configuration shown in 'Image 8.3', when a new screen is added, it should just fit into the empty slot	All screens shrink in size when a new screen is added, as shown in 'Image 8.4'	Reason for this error is due to the total number of slots only incrementing by 1 when the tileLength shrinks enough to create space for new new slots. I did not take into consideration multiple layers, each having an additional slot
8D	Testing the improved best-fit picturebox algorithm when adding screens	Normal	Same as 8C	As expected	Pass

8E	Testing that changes to the tile map done in one screen applies to all other screens in pnlGrid	Normal	All screens should be updated the same way as the edited screen	See ‘Image 8.5’	Pass
8F	Testing that path points placed in one screen will also be placed in all other screens	Normal	All screen overlays should be updated the same way as the edited screen	See ‘Image 8.6’	Path points only appear in Screen0, even when clicking on other screens, but it can be deleted from other screens. Solved by drawing and removing path points to all overlays as a path point is placed and removed
8G	Testing that path points placed in one screen will also be placed in all other screens	Normal	All screen overlays should be updated the same way as the edited screen	See ‘Image 8.7’	Pass
8H	Testing that the improved best-fit picturebox algorithm works when there are multiple empty slots to begin with	Extreme	From ‘Image 8.8’, new screens should just fill in the available slots	See ‘Image 8.9’	Tile length increases after adding a new screen, causing the two screens to be too large for pnlGrid to display fully. This is solved by correcting the initial values of totNoOfSlots, slotsPerLayer, noOfLayer
8I	Same as 8H	Extreme	Same as 8H	See ‘Image 8.10’	Pass

8J	Testing that the improved best-fit picturebox algorithm keeps the tileLength greater than or equal to the minimum tileLength	Extreme	The algorithm should reject a screen state change if it causes the tileLength to go below a minimum value	See ‘Image 8.11’	TileLength goes below minimum (Shown by the black tile in ‘Image 8.11’, which should be an ‘X’) Later I found that everything was actually working correctly. The minimum tileLength was set to 3 pixels, which includes the pixel used as a grid line, resulting in a 2x2 pixel area for the ‘X’

Images

Image 8.1:

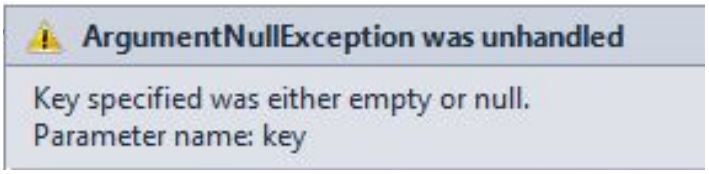


Image 8.2:

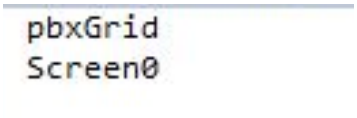


Image 8.3:

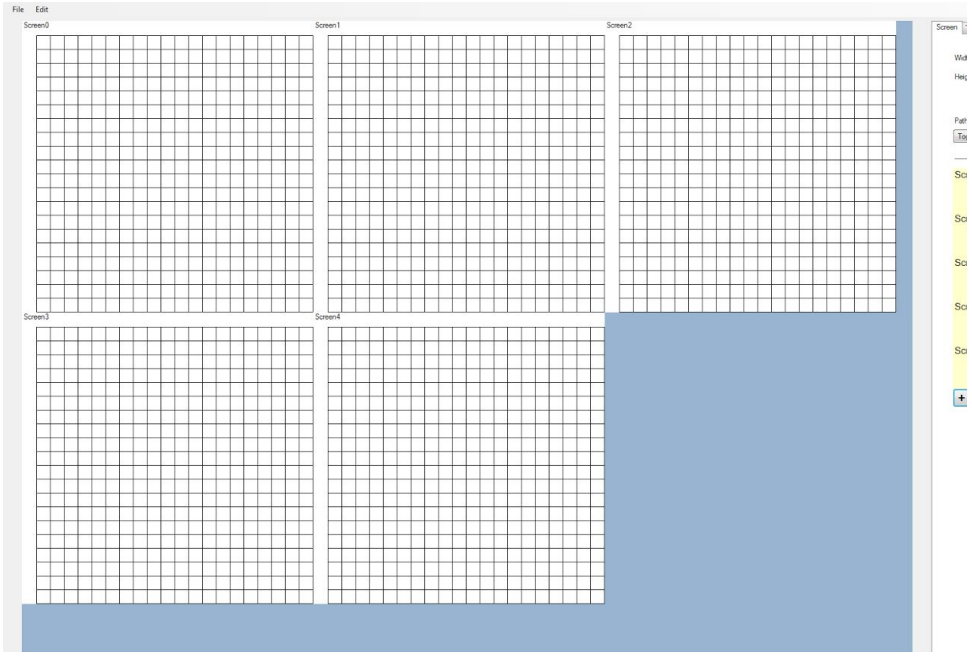


Image 8.4:

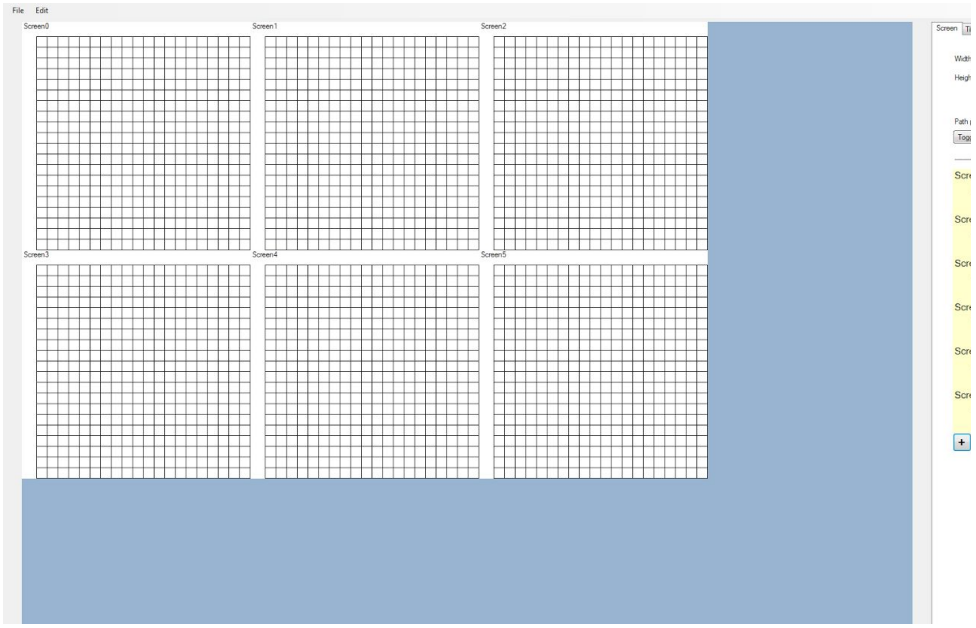


Image 8.5:

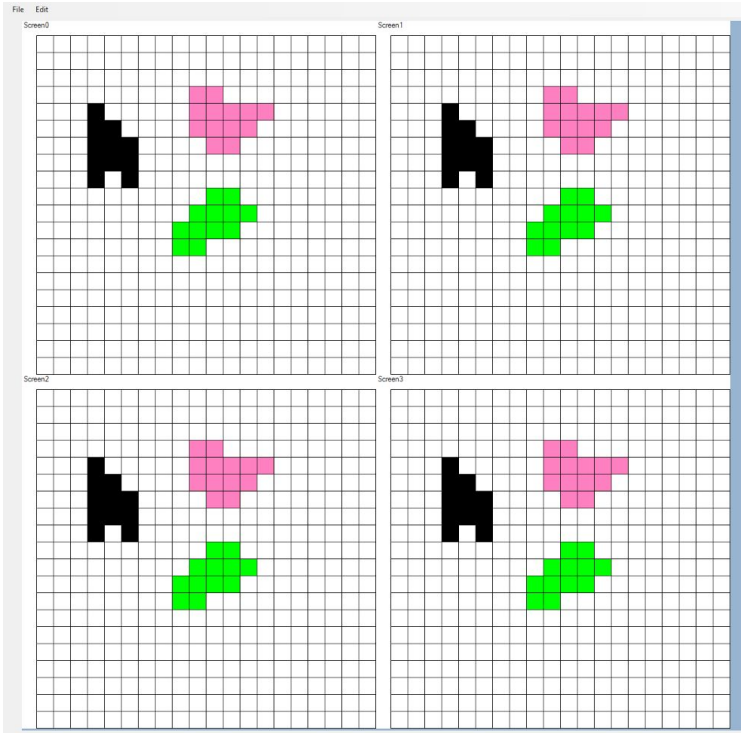


Image 8.6:

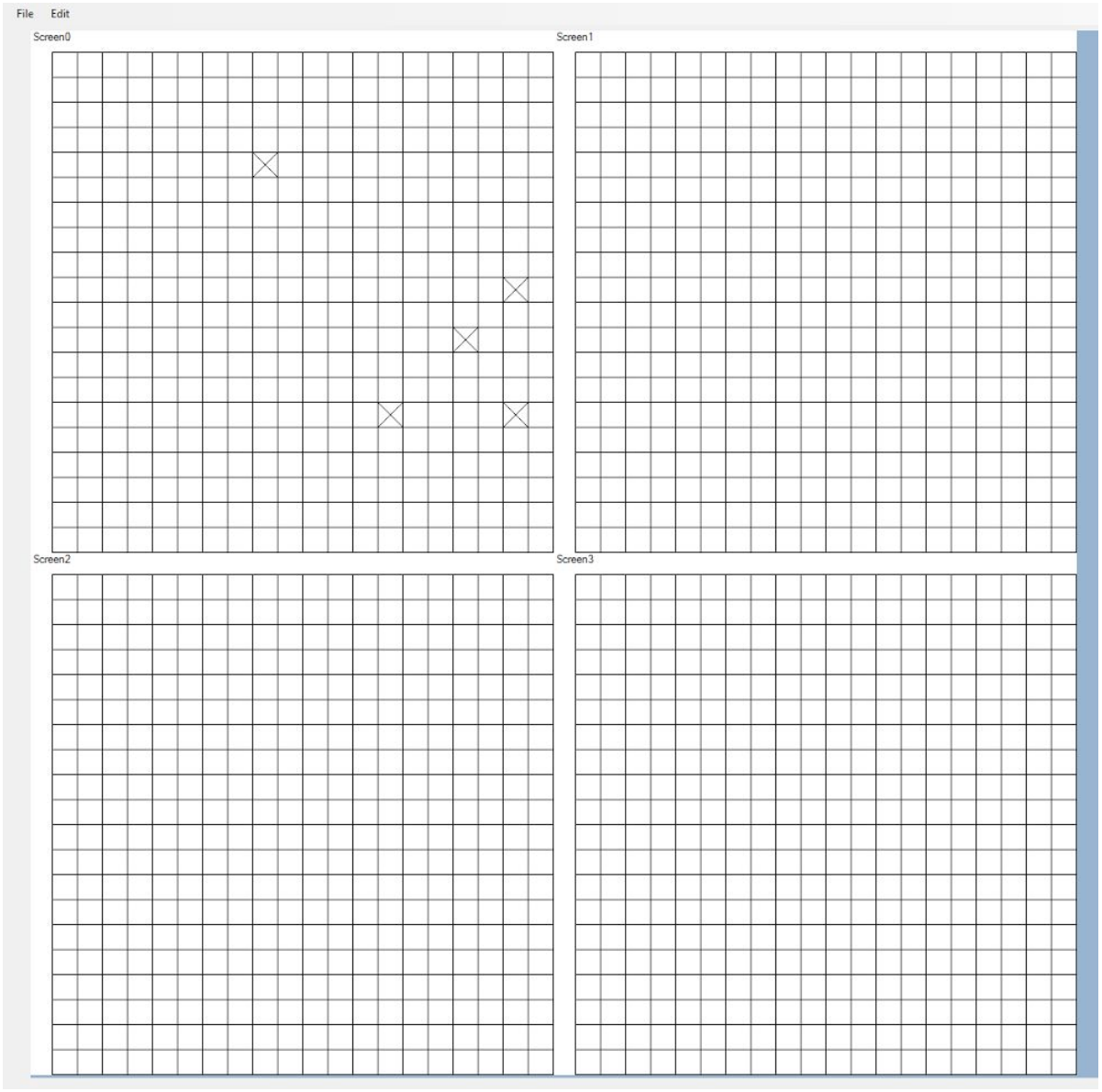


Image 8.7:

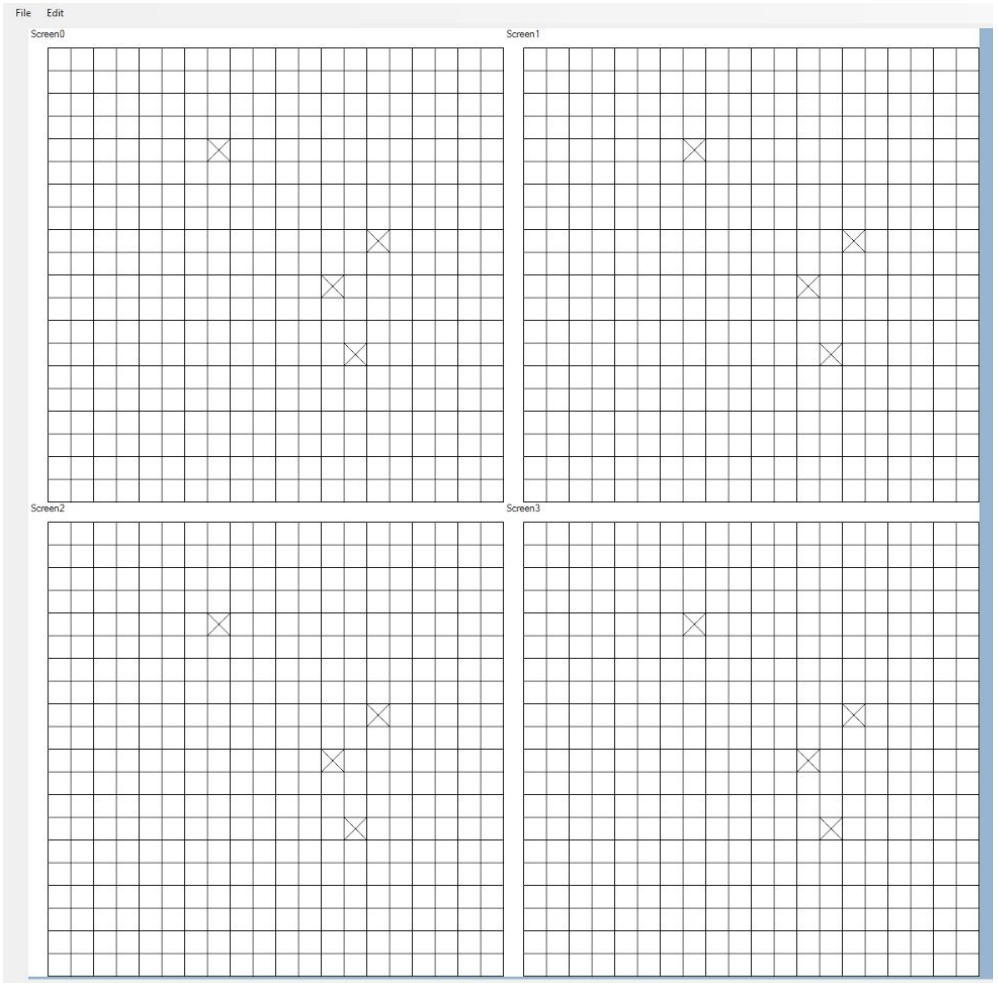


Image 8.8:

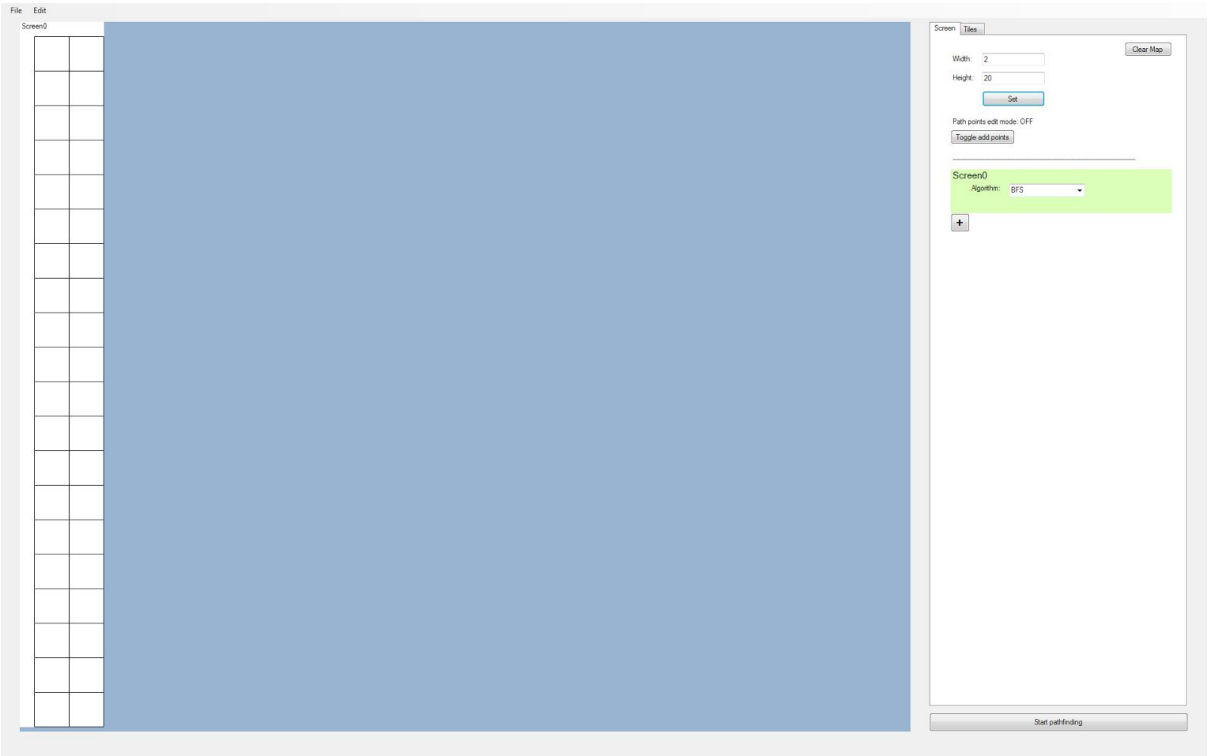


Image 8.9:

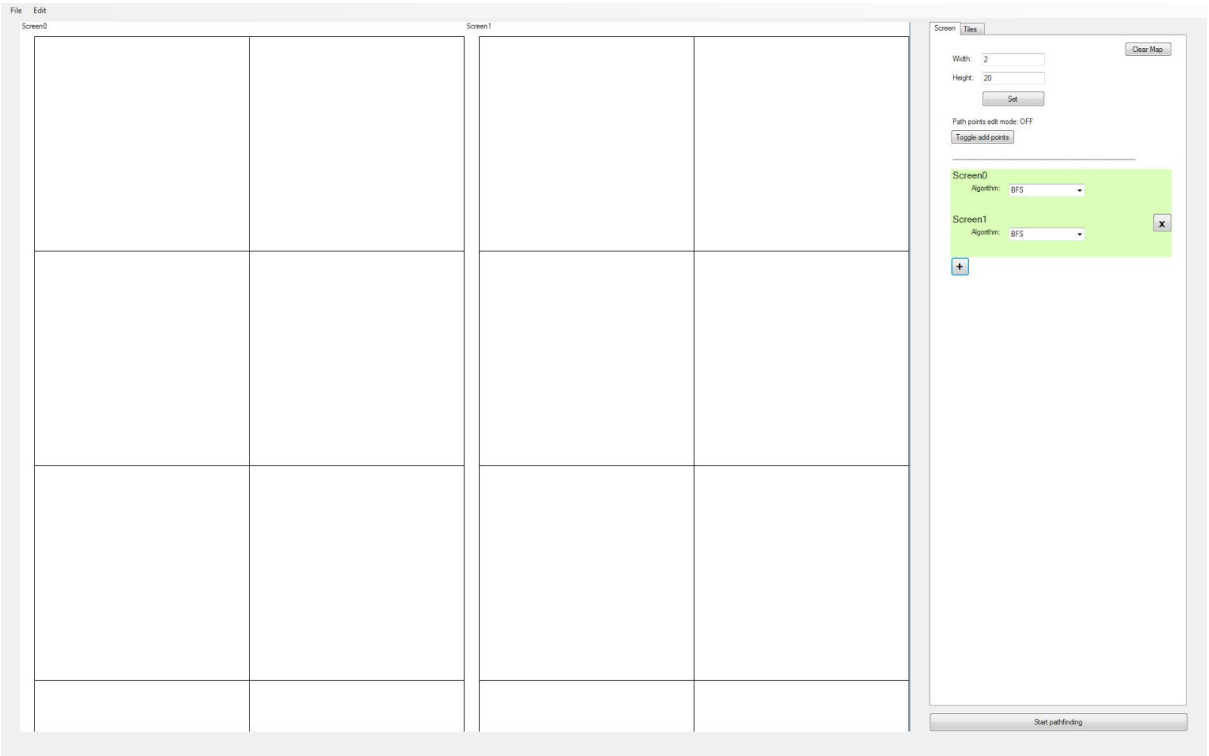


Image 8.10:

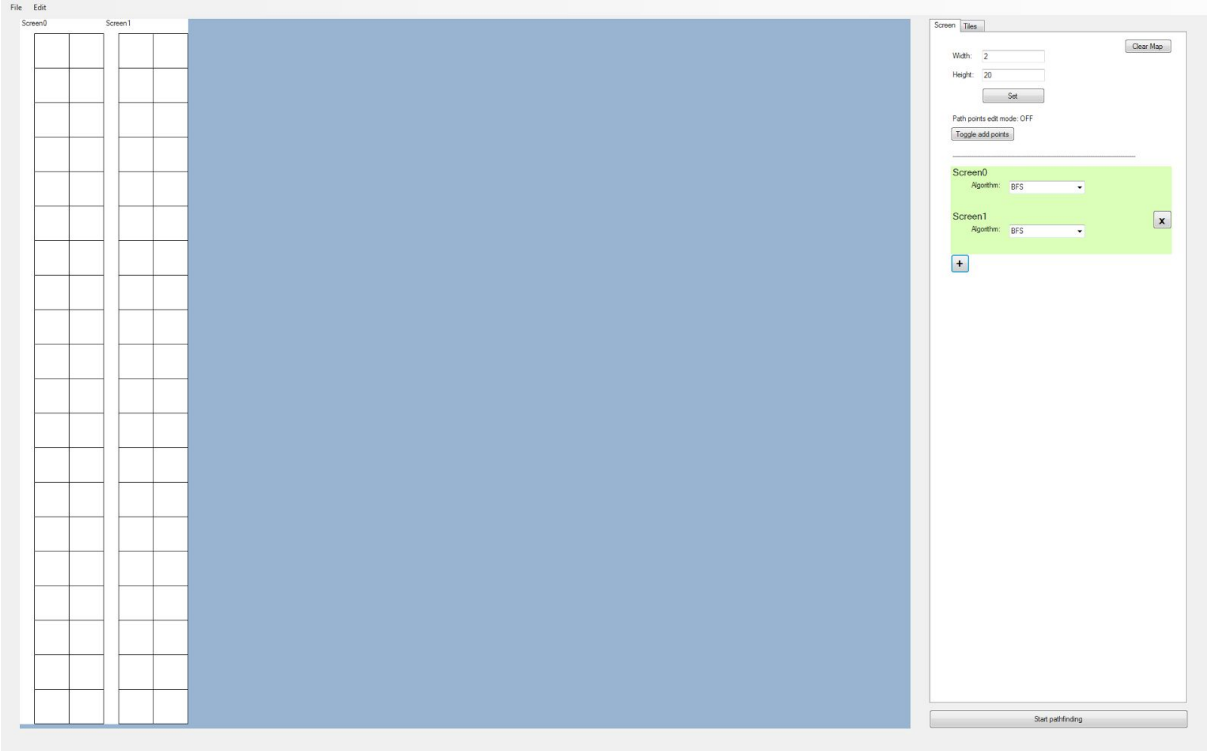
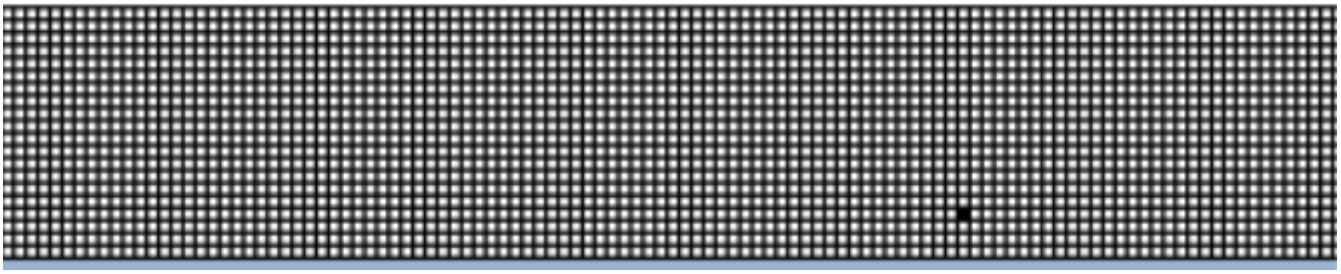


Image 8.11:



9. Dijkstra's Algorithm

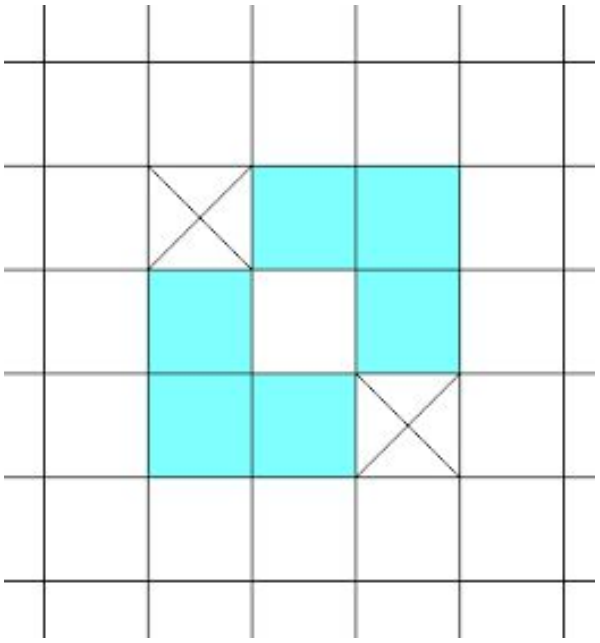
Objective: The user should be able to pathfind between the set path points using Dijkstra's Algorithm.

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
9A	From the tile map configuration shown in 'Image 9.1', where white tiles have a weight of 1, and blue tiles have a weight of 2, the algorithm should find the shortest path from the source to the destination node	Normal	See 'Image 9.2'	See 'Image 9.3'	The expected path had a distance 6, while the actual path calculated via Dijkstra's algorithm had a distance 8. Clearly, this is not the shortest path
9B	Debugging test 9A	Normal	See 'Image 9.4'	As expected	Through self-calculations, the calculated tentative distances of each tile appear to be correct. The reason why a suboptimal path was drawn is because the initial method of backtracking to the source tile is flawed
9C	Obtaining shortest path with new backtracking algorithm	Normal	See 'Image 9.5'	As expected	Pass
9D	Comparing BFS with Dijkstra's algorithm	Normal	See 'Image 9.6' and 'Image 9.7'	As expected	Although the path taken is different, the total cost of each path is the same. Pass

9E	Testing Dijkstra's algorithm with a weighted map - See 'Image 9.8'	Normal	See 'Image 9.9'	See 'Image 9.10'	The actual solution has a weight of 140, which is the same as the expected weight. Like the expected solution, the actual solution goes through the least number of tiles in the heavier weighted blue section of the map. Pass

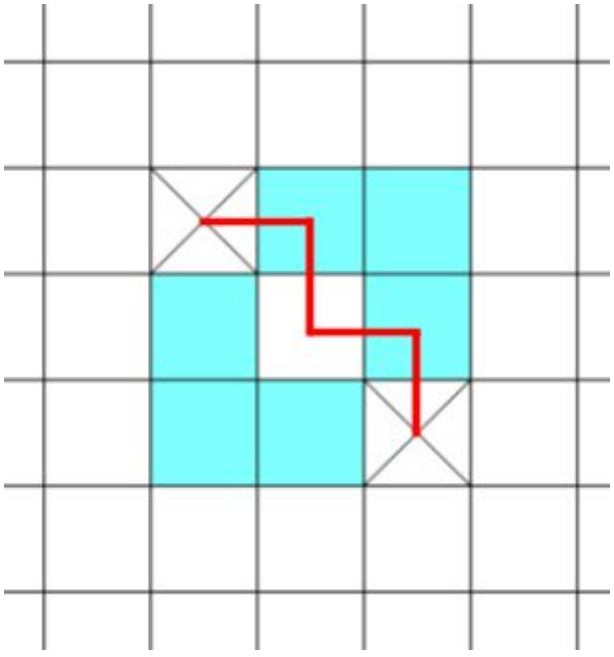
Images

Image 9.1:



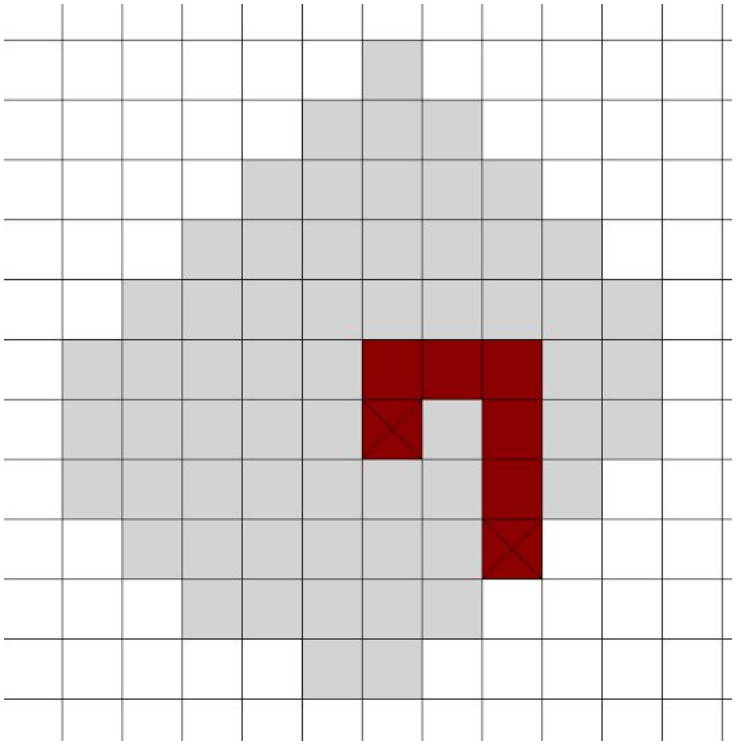
(Weights: White = 1, Blue = 2)

Image 9.2:



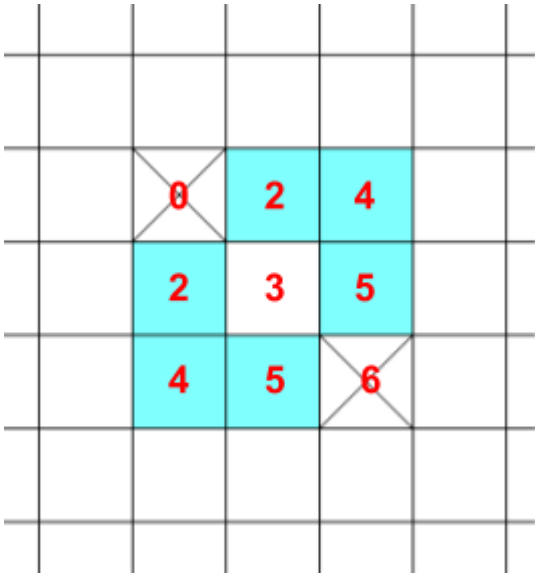
(Expected path, distance 6)

Image 9.3:



(Actual path, distance 8)

Image 9.4:



(Distance from start node (Top left))

Image 9.5:

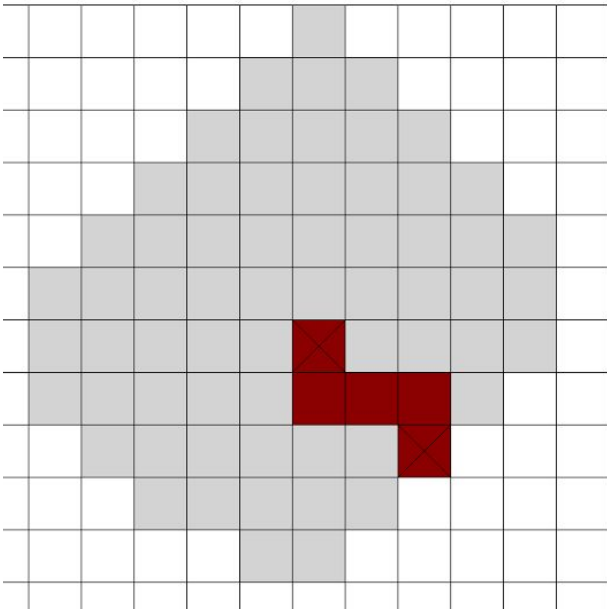
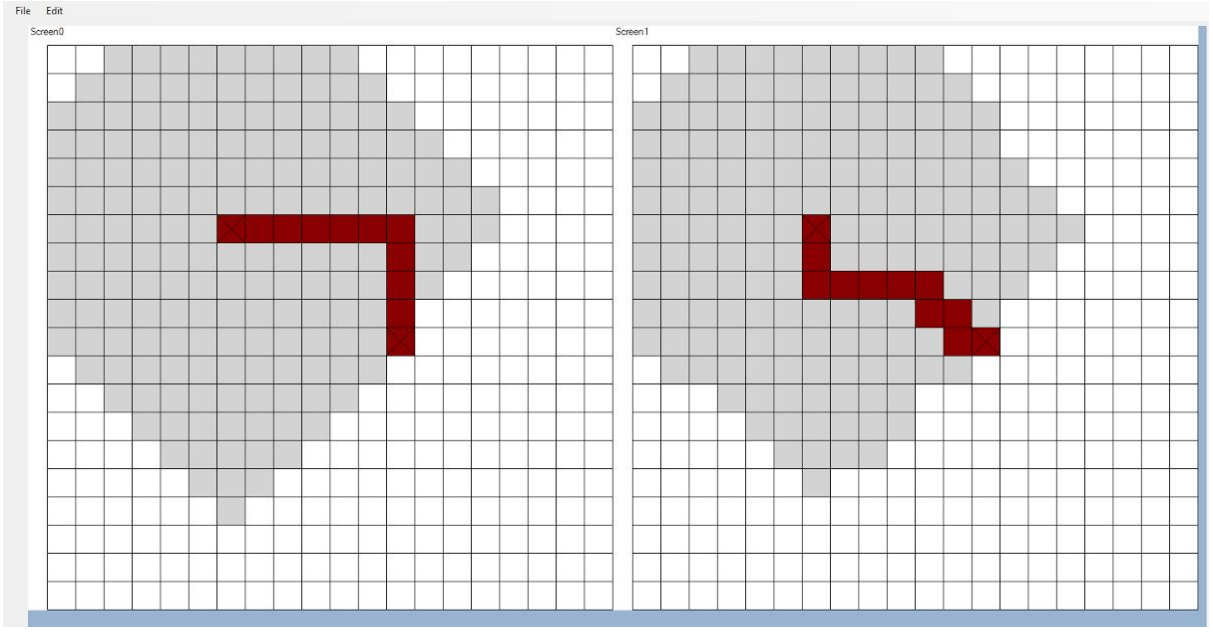
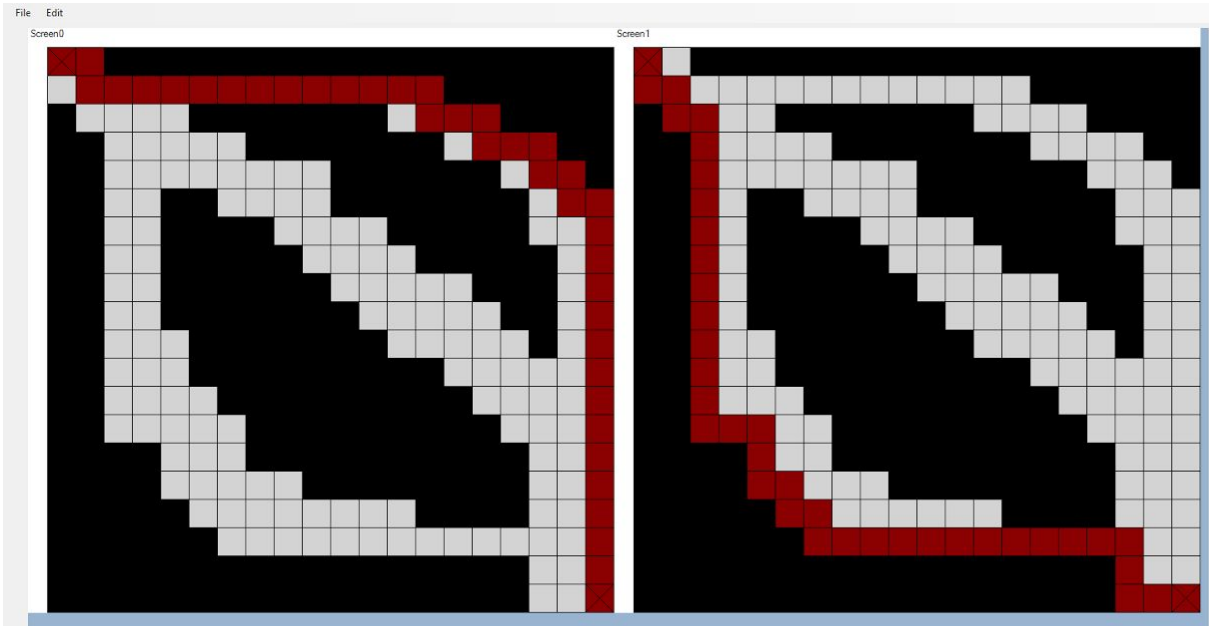


Image 9.6:



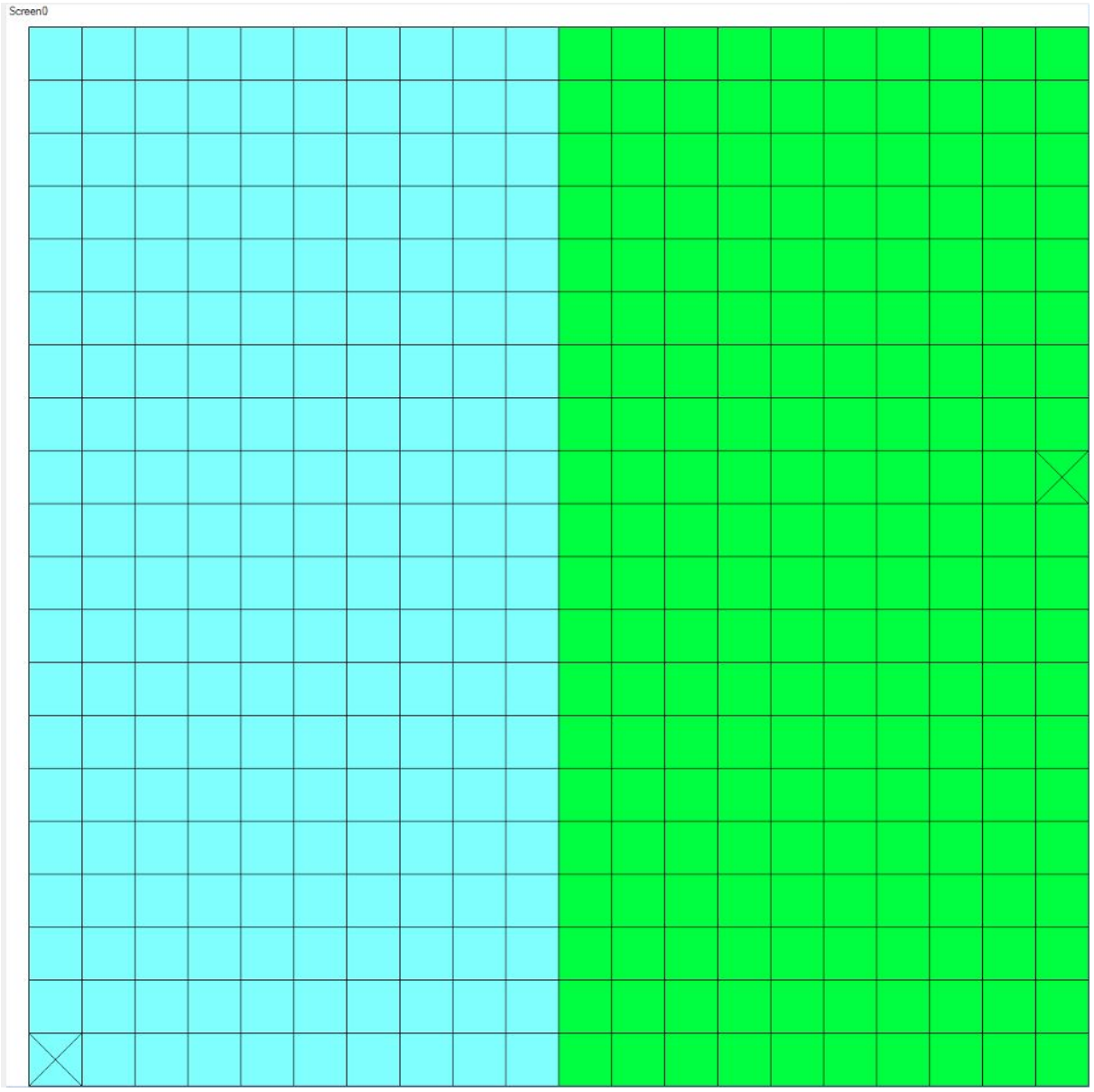
Left: BFS, Right: Dijkstra

Image 9.7:



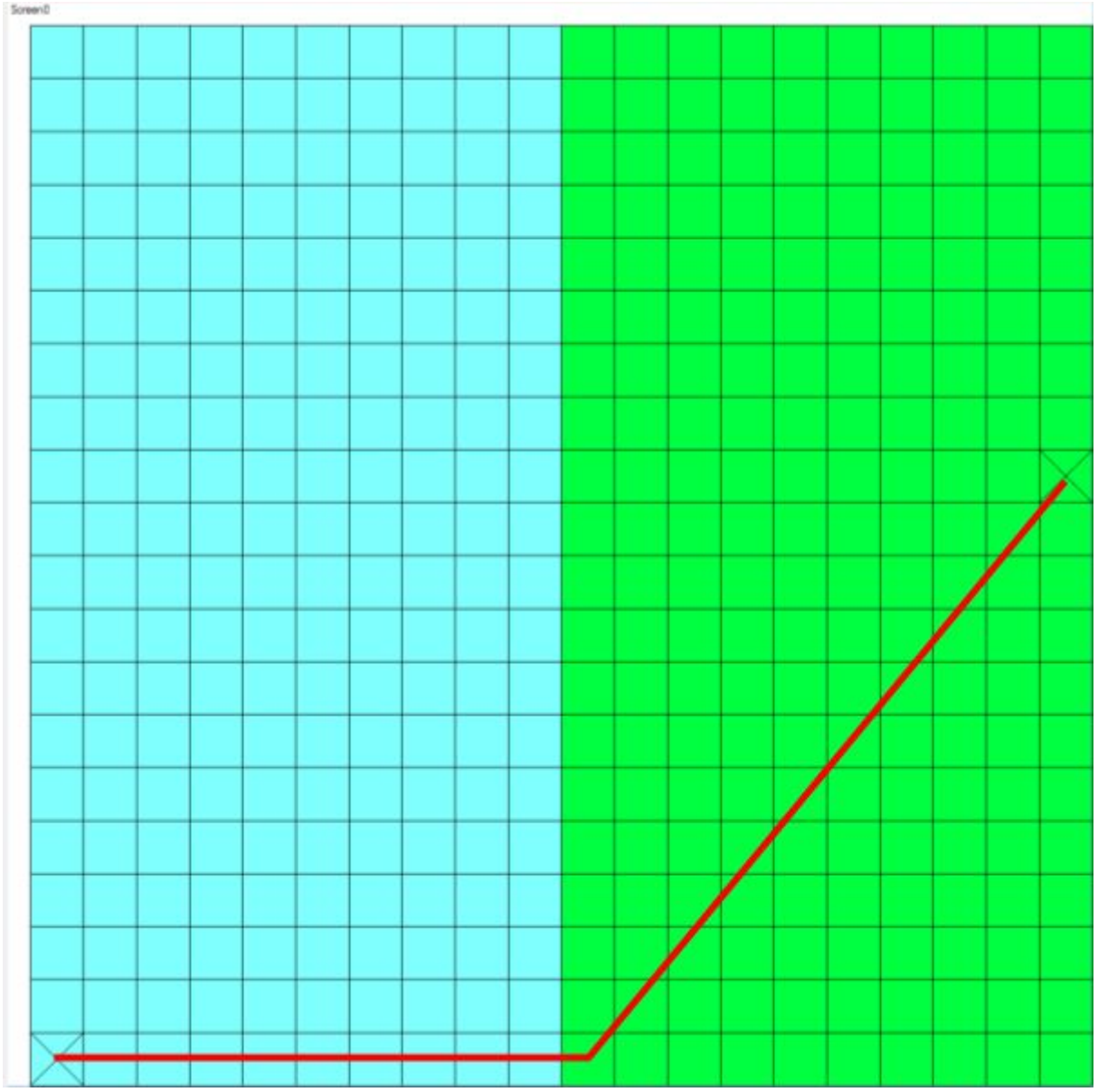
Left: BFS, Right: Dijkstra

Image 9.8:



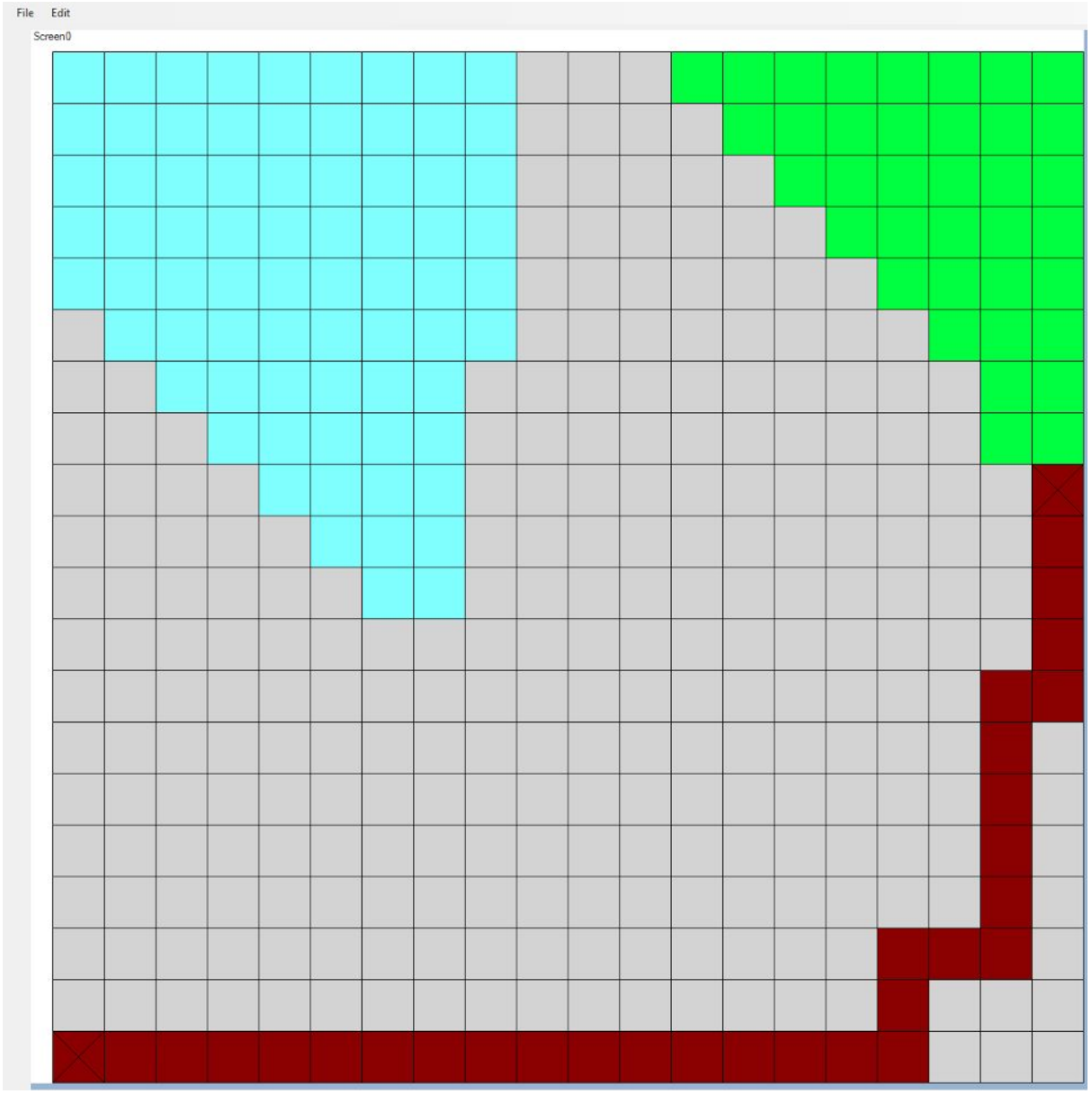
(Weights: Blue = 10, Green = 2)

Image 9.9:



(Weights: Blue = 10, Green = 2)

Image 9.10:



(Weights: Blue = 10, Green = 2)

10. A* Algorithm

Objective: The user should be able to pathfind between the set path points using the A Algorithm. With this algorithm selected, the user should also be able to select between the 'Manhattan', 'Euclidean', and 'Chebyshev' heuristics, which will alter the way the algorithm searches for the shortest path.*

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
10A	Testing the A* algorithm with different heuristic settings	Normal	The search areas for each variation of the algorithm should show that the algorithm is looking at tiles that are towards the destination point	The Manhattan and Chebyshev heuristics have the expected search area, but Euclidean seems to have the same search area shape as Dijkstra's algorithm, seen in 'Image 10.1'	The Euclidean heuristic seems to be searching away from the destination point
10B	Debugging the A* Euclidean heuristic setting on a simple 3x3 map	Normal	There should be a small search area when looking towards the destination point	See 'Image 10.2'	Further analysis of the situation, shown in 'Image 10.3', shows that there is already an issue in the first iteration - The expected H score of 2nd line is 4, but the actual value recorded is 2. The reason for this is due to the misuse of the '^' operator in C# used to calculate the H-Score, which does not stand for exponentiation.

10C	Debugging the A* Euclidean heuristic setting on a simple 3x3 map	Normal	See ‘Image 10.4’ and ‘Image 10.5’	As expected	Pass
10D	Testing A* algorithms through a test map and comparing their shortest paths with BFS’s shortest path	Normal	The A* algorithms should all match the path taken by the BFS algorithm	See ‘Image 10.6’	Both the Manhattan and Euclidean heuristics are not finding the shortest path. Both heuristics are admissible for this type of map so there must be something wrong with the A* algorithm
10E	Debugging A* algorithm	Normal	See ‘Image 10.7’	See ‘Image 10.8’	The actual F-Scores of each tile do not match what is expected. The trace table in ‘Image 10.9’ shows the issue - The tilesToCheck dictionary does not reorder itself after ‘dequeuing’ a tile. Tile 5,2 should have been checked next, but due to the empty slot above it, tile 5,3 gets placed there and gets checked next, causing the change in pattern observed in the ‘hallway’ in ‘Image 10.8’ and thus causing incorrect F-Scores
10F	Debugging A* Algorithm with new custom priority list’	Normal	See ‘Image 10.10’	See ‘Image 10.10’	Pass

Images

Image 10.1:

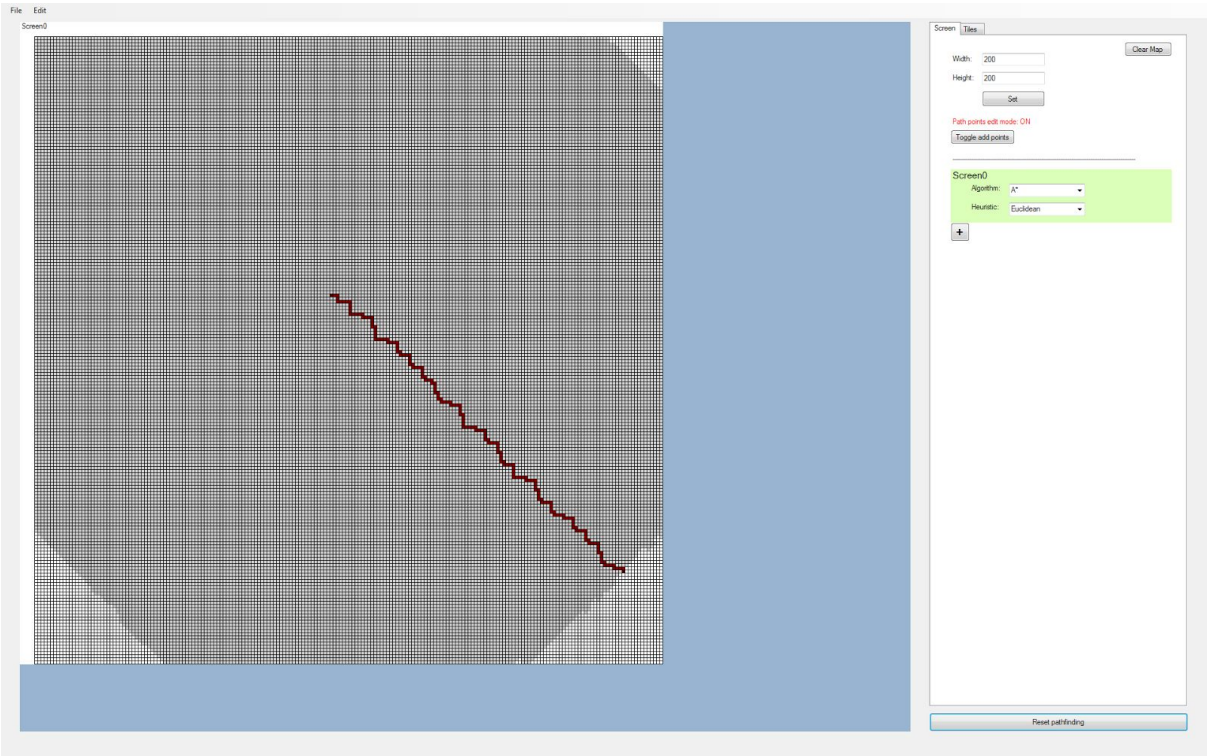
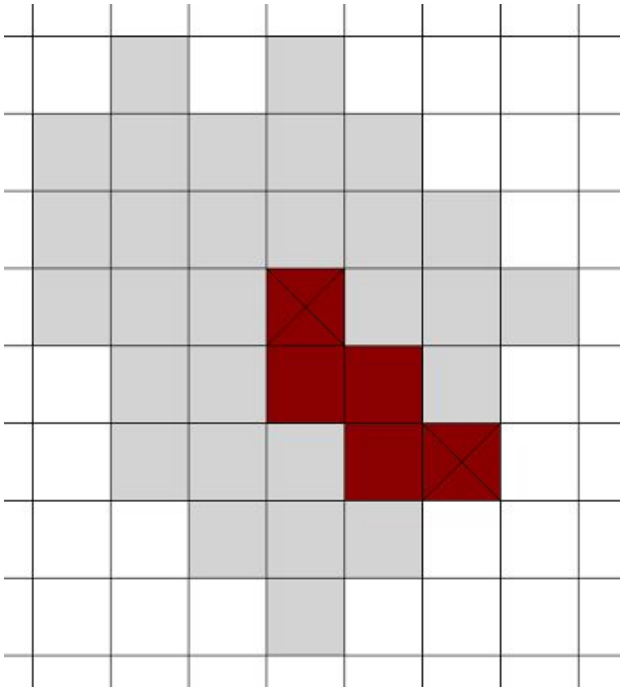


Image 10.2:



(A* Euclidean, from 10,10 to 12, 12)

Image 10.3:

Dequeued 10,10, G: 0, H: 0, F: 0
Added 10,9, G: 1, H: 2, F: 3
Added 9,10, G: 1, H: 2, F: 3
Added 10,11, G: 1, H: 1, F: 2
Added 11,10, G: 1, H: 2, F: 3

(Debug Output)

Image 10.4:

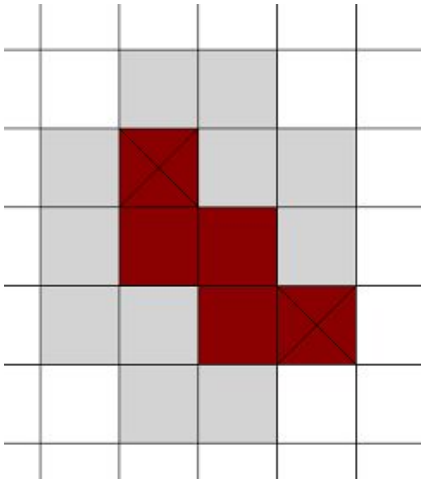
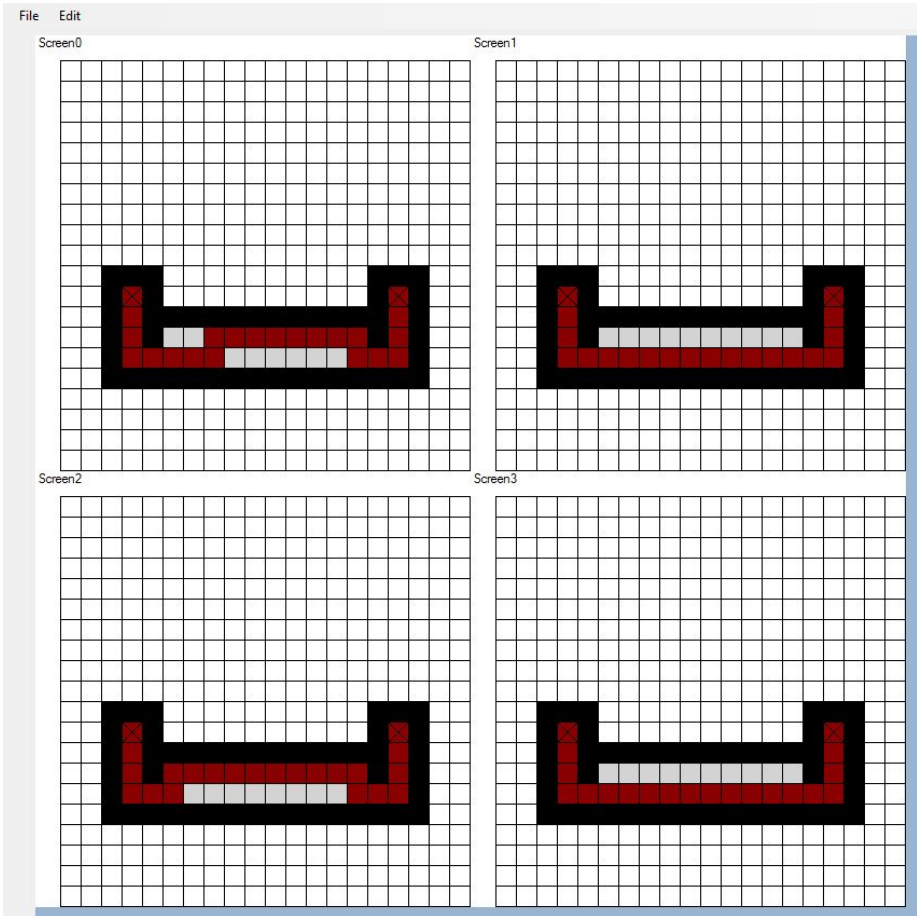


Image 10.5:

Dequeued 10,10, G: 0, H: 0, F: 0
Added 10,9, G: 1, H: 4, F: 5
Added 9,10, G: 1, H: 4, F: 5
Added 10,11, G: 1, H: 2, F: 3
Added 11,10, G: 1, H: 2, F: 3

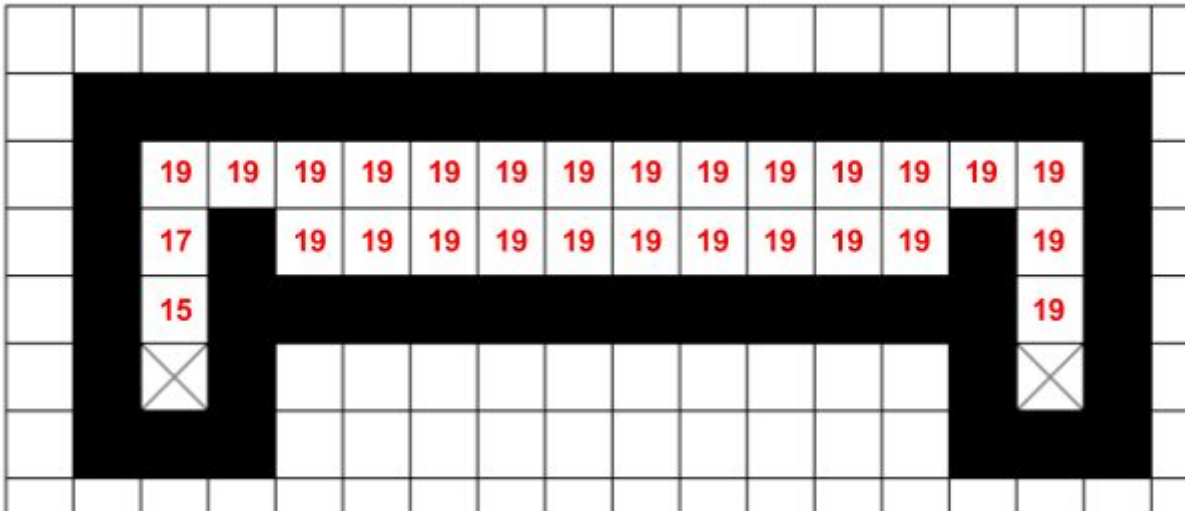
(Debug Output)

Image 10.6:



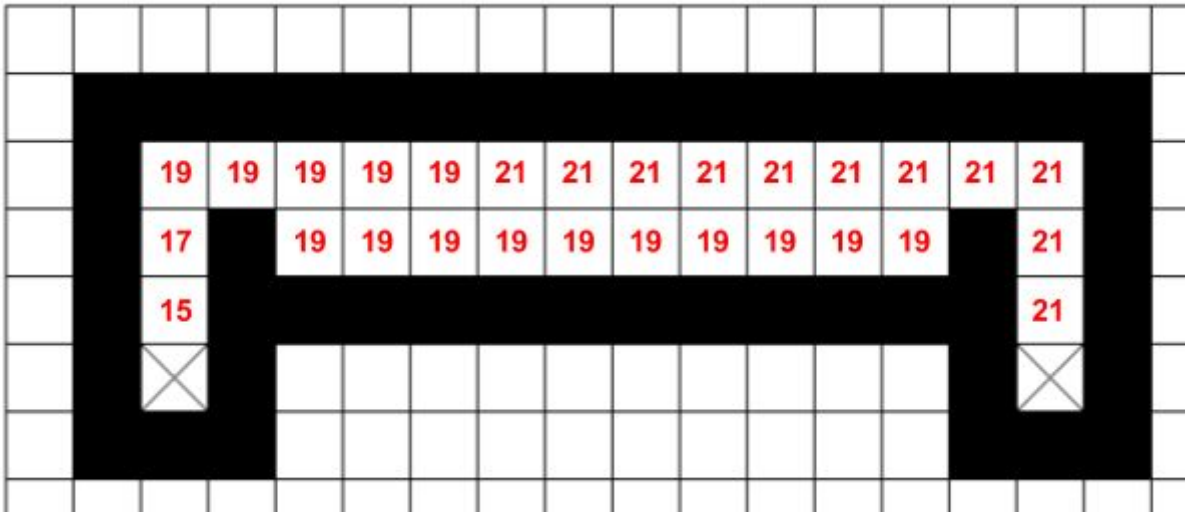
Top left = A* Euclidean
Top Right = BFS
Btm Left = A* Manhattan
Btm Right = A* Chebyshev

Image 10.7:



(Expected F-Scores)

Image 10.8:

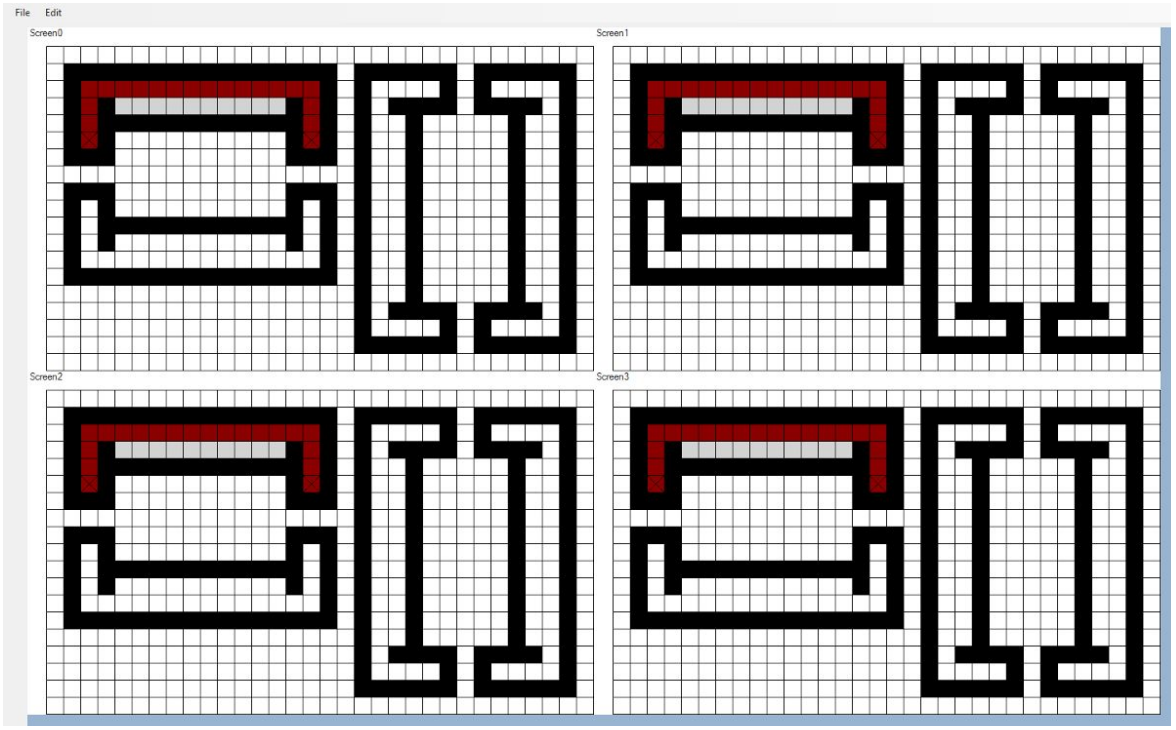


(Actual F-Scores)

Image 10.9:

currTile	nextTile	tilesToCheck
3,2	3,1	-
	2,2	-
	3,3	-
	4,2	4,2, F:19
4,2	4,1	-
	3,2	-
	4,3	4,3, F:19
	5,2	4,3, F:19 5,2, F:19
4,3	4,2	- 5,2, F:19
	3,3	- 5,2, F:19
	4,4	- 5,2, F:19
	5,3	5,3, F:19 5,2, F:19
5,3, F:19	5,2	- 5,2, F:19

Image 10.10:



(All algorithms pass all 8 tests. Only one of these tests is shown in the above image)

11. Exporting Map and Importing Images

Objective: The user should be able to export the map as a bitmap file, where each pixel represents a single tile. The user should also be able to import images into the program.

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
11A	Exporting a 23 x 7 test image shown in ‘Image 11.1’	Normal	See ‘Image 11.3’	See ‘Image 11.2’	Exporting works, but rather than exporting the image as it is displayed on the screen, I want to export a bitmap where each pixel represents a tile in the grid
11B	Exporting a 23 x 7 test image shown in ‘Image 11.1’	Normal	See ‘Image 11.3’	Access Violation Exception thrown	When drawing the bitmap to export, I thought that the pointer points to the bits of the bitmap so I multiplied by 32 to move to the next pixel. In actuality, the pointer points to bytes. Multiplying by 4 instead solves the problem
11C	Exporting a 23 x 7 test image shown in ‘Image 11.1’	Normal	See ‘Image 11.3’	As expected	Pass
11D	Exporting a coloured version of the 23 x 7 test image shown in ‘Image 11.1’	Normal	See ‘Image 11.4’	As expected	Pass

11E	Importing an image to the program, where a tileType is created for each unique colour in that image	Normal	The program imports the image correctly and creates a tileType for each unique colour in that image	Dictionary key not found exception thrown when adding coordinates into a HashSet stored as a dictionary value	The error is due to me not taking into account the keys of the default tileTypes and treated their keys the same way as the custom tileTypes. So when it came to adding default tiles to the HashSet, I searched for a key that did not exist rather than searching for 'Empty' or 'Wall'
11F	Same as 11E	Normal	Same as 11E	See 'Image 11.5'	Pass
11G	Importing the same image as in 11E, but with a map size that is different from the image size	Normal	The program should resize the map to the image size and import it correctly	Access Violation Exception thrown when filling colour list and looping through bitmap data	This error is due to the method 'fillColourList' using the current number of tiles in the for loop rather than the number of tiles of the image to import. This will either cause the pointer to overshoot or undershoot the bitmap in system memory. Altering the for loop stopping conditions to the bitmap width and height resolves this issue
11H	Same as 11G	Normal	Same as 11G	Index Out Of Range Exception thrown when attempting to output image to map on accessing lstMapRow	The global variable that stores the number of tiles in X and Y did not update after the map was rebuilt. The for loop used to output the image to the map depends on these two values. Thus, causing an index out of range error. Issue resolved by updating Global.NoOfTilesXY after rebuilding map

11I	Setting new tileType values on the tileTypes of the newly imported image	Normal	The tileTypes should update once valid values have been entered and set	Index Out Of Range Exception thrown when btnSet is clicked	Debugging the problem shows that there is a comma missing in the coordiantes: tileCoords = “11”. This problem is caused due to inserting coordinates into the hashSet without the separator character, thus causing the problem
11J	Same as 11I, but with comma added into the coordinates in the HashSet	Normal	Same as 11I	See ‘Image 11.6’	Pass
11K	Testing that if we start off with multiple screens, importing will add the test image to all 10 screens	Normal	All 10 screens should display the imported image	See ‘Image 11.7’	Pass
11L	Testing that when importing the image shown in ‘Image 11.8’, the program will reject importing the image when there are too many screens (In this case, 7 screens)	Erroneous	The program should reject importing the image	See ‘Image 11.9’	Pass
11M	Testing that the program does not accept importing non-images into the program	Erroneous	The program should reject importing the non-image	See ‘Image 11.10’	Pass
11N	The maximum number of unique colours is temporarily changed to 2. When importing an image with 3 colours, the program should reject it	Erroneous	The program should reject importing the image	See ‘Image 11.11’	Pass

Images

Image 11.1:

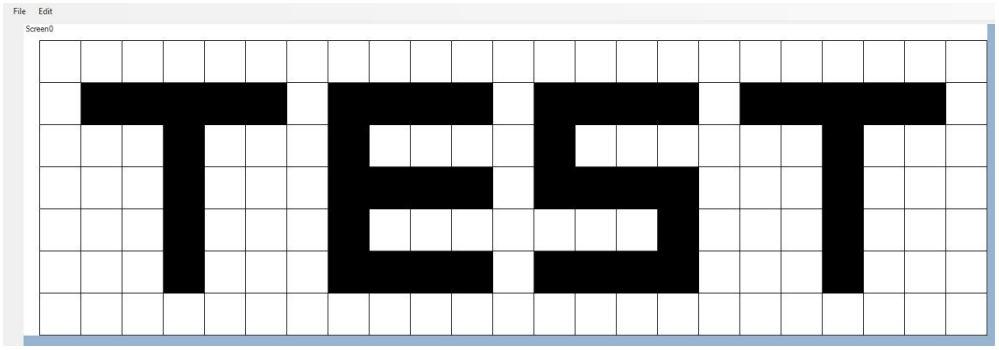


Image 11.2:

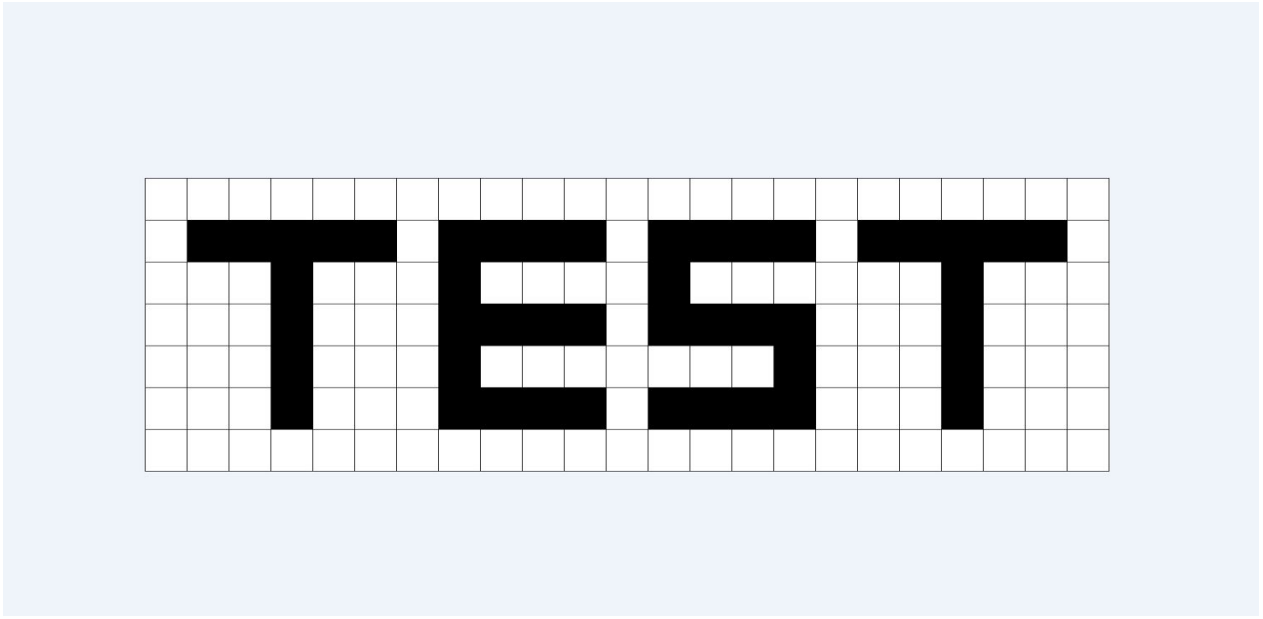


Image 11.3:

(Zoomed in)

Image 11.4:

(Zoomed in)

Image 11.5:

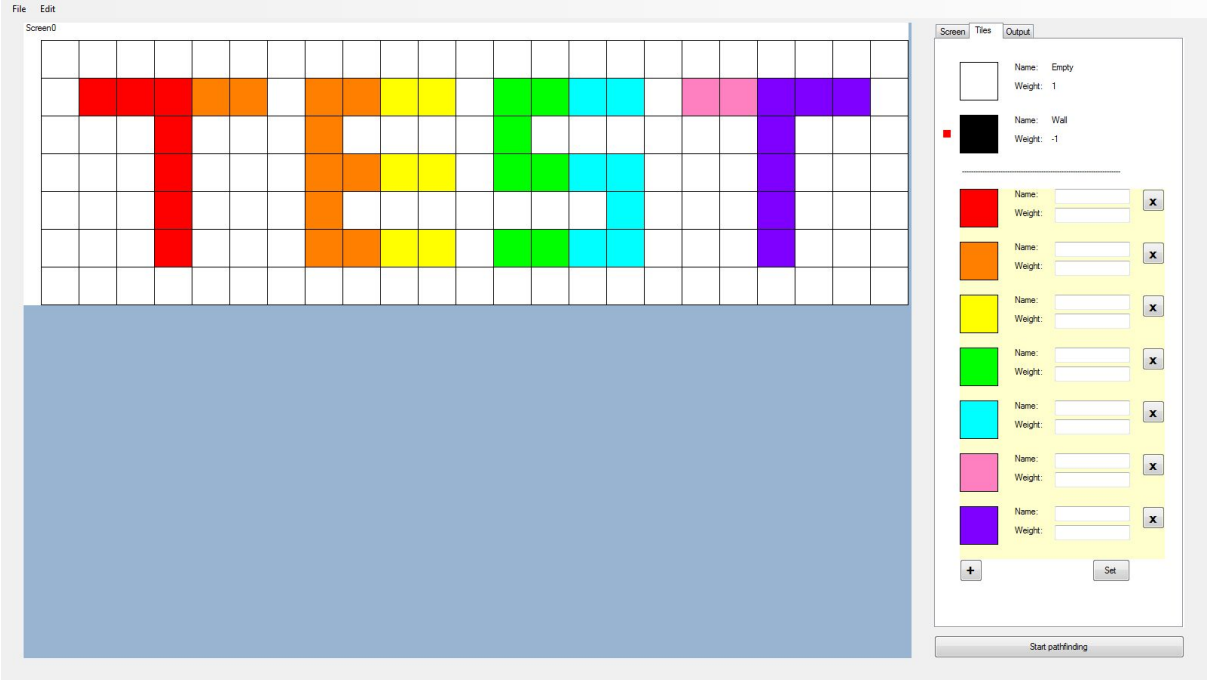


Image 11.6:


Name: Empty
Weight: 1
Name: Wall
Weight: -1
Name: a
Weight: 1
Name: b
Weight: 2
Name: c
Weight: 3
Name: d
Weight: 4
Name: e
Weight: 5
Name: f
Weight: 6
Name: g
Weight: 7
x
+
Set

Image 11.7:

Image 11.8:

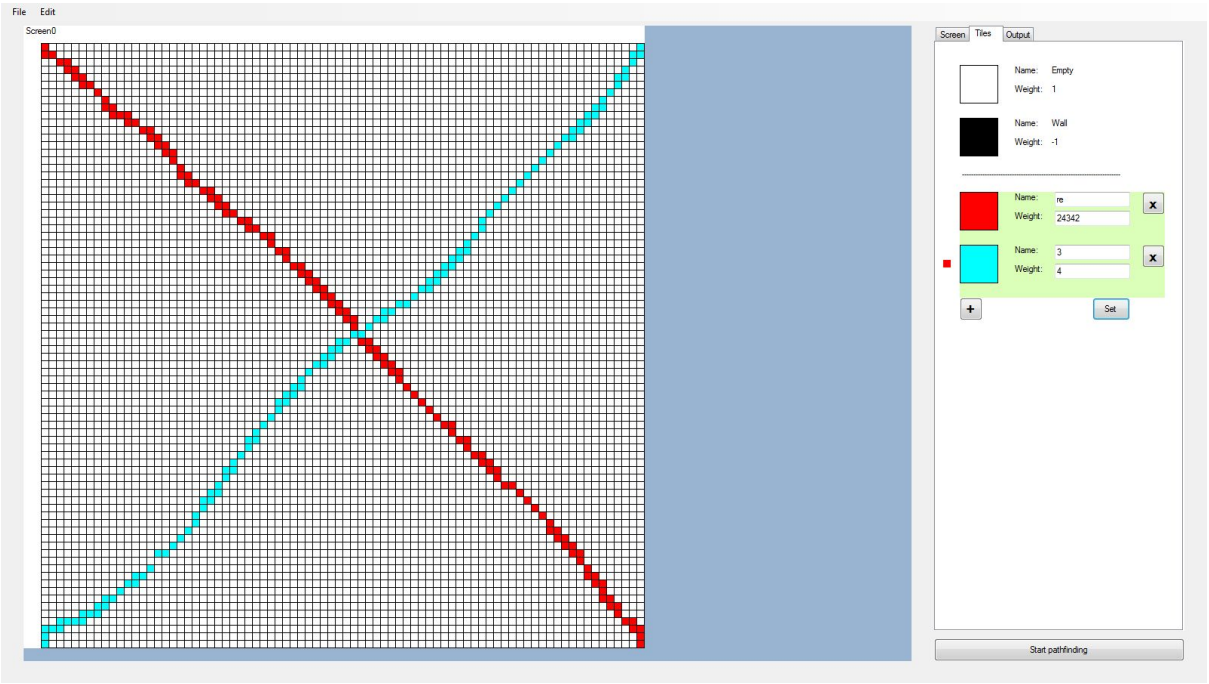


Image 11.9:

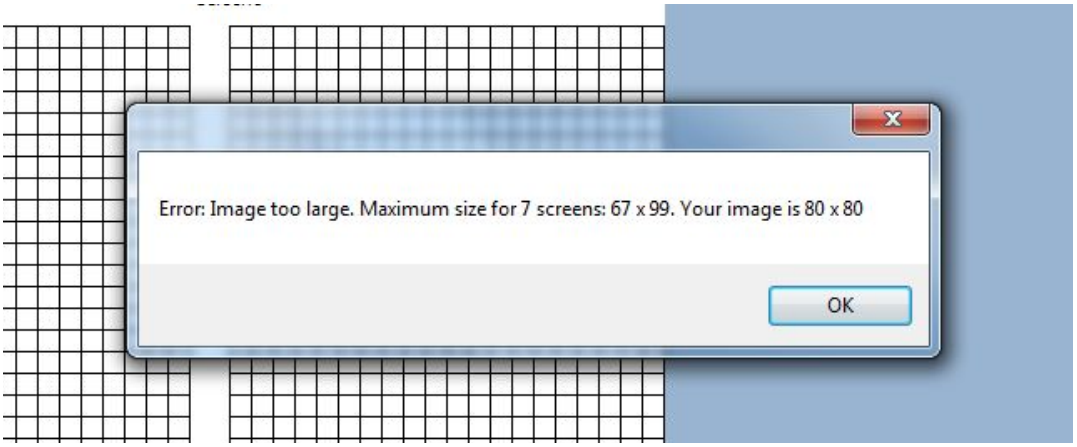


Image 11.10:

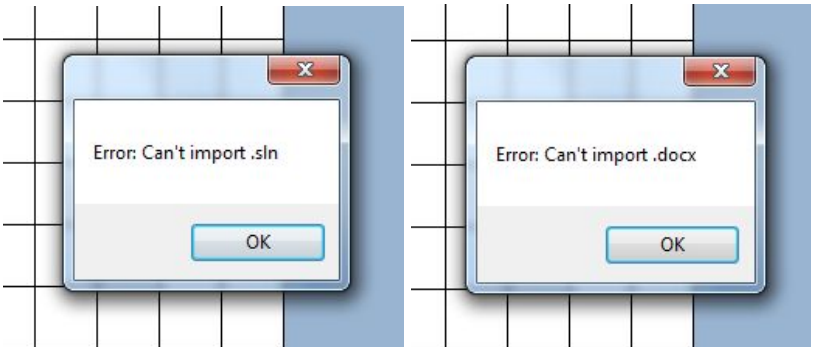
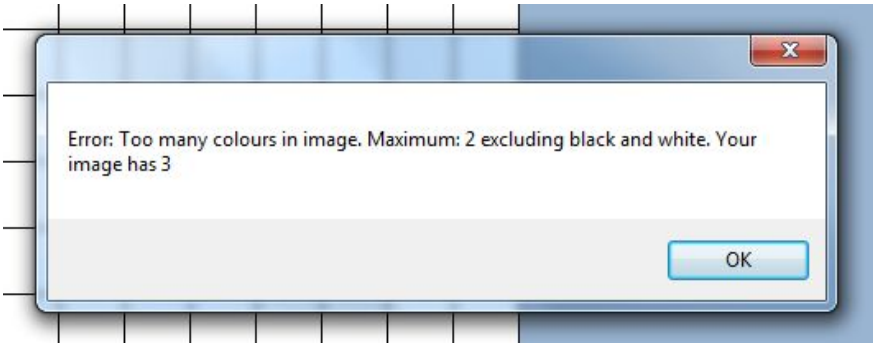


Image 11.11:



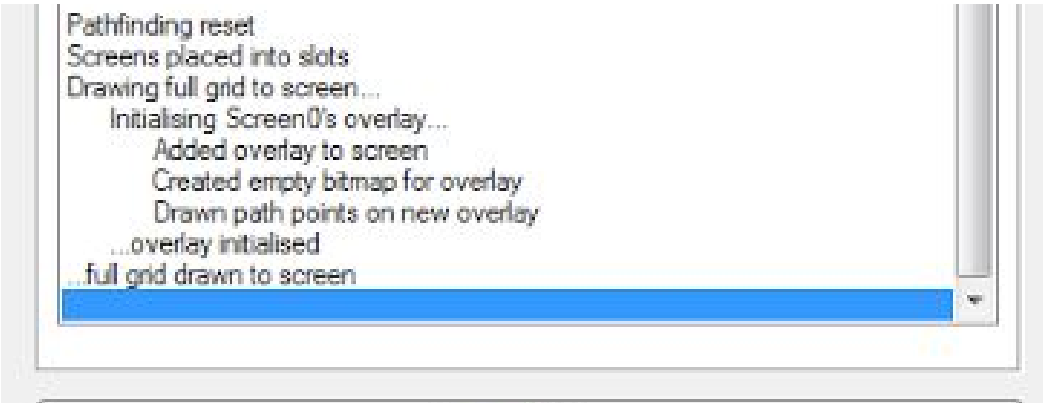
12. Outputting Program Status

Objective: The user should be able to view the state of the program as it runs. After pathfinding, the user should also be able to compare the algorithms by looking at the number of steps and time taken to find the shortest path.

Test No.	Description	Data Type	Expected Result	Actual Result	Comments
12A	Testing that when the ‘Start Pathfinding’ button is clicked, the selected tab should switch to tabOutput, which houses the program log output, lstOutput	Normal	The selected tab should switch to tabOutput	As expected, but the text does not display immediately	Solved by refreshing tabOutput after switching to it
12B	Same as 12A, but testing that lstOutput scrolls to the bottom upon switching to tabOutput	Normal	Same as 12A, but lstOutput should scroll to the bottom upon switching tabs	As expected, but after scrolling down, a highlight on lstOutput, as shown in ‘Image 12.1’, is visible	Attempted to solve the problem by disabling lstOutput
12C	Same as 12B	Normal	Same as 12B	As expected, but it is not possible to scroll back up	Solved by continuously deleting messages from the top until the scrollbar is not visible

Images

Image 12.1:



Evaluation

General Overview

The final outcome largely meets the program requirements as the user can create a custom map, or import a pre-made map, create copies of it, and run different pathfinding algorithms for each copy. Once all pathfinding algorithms have been executed, the times taken for the algorithm to find the shortest path is output and this can be used to compare the different pathfinding algorithms the user has selected.

Objective Evaluation

Objective	Met?	Comments
The program should display a 2D grid of tiles with a valid ‘size’ (Number of tiles in a row and column) defined by the user.	Yes	-
The user should be able to create their own obstacles with custom name, weight, and colour.	Yes	Although the user is able to create custom tileTypes of their own, I feel that the code would be more organised if I had created classes for the tileType panels rather than leaving it as a subroutine
The user should be able to add obstacles of their choice onto the grid.	Yes	-
The user should be able to clear the map.	Yes	-
The user should be able to navigate through the map if it does not all fit into the screen.	No	This objective is obsolete as it has been decided that all screens must fit into pnlGrid, given that the map size is small enough. Thus, there is no need to zoom or pan the map

The user should be able to choose between at least 3 different pathfinding algorithms.	Yes	The user is able to choose from 5 different algorithms: BFS Dijkstra A* - Manhattan A* - Euclidean A* - Chebyshev
For all pathfinding algorithms, the shortest path from the starting tile to the destination tile should be displayed after it has been found.	Yes	-
The user should be able to compare multiple pathfinding algorithms of their choice by displaying copies of the map, each with a different pathfinding algorithm running. By pressing ‘play’, all chosen pathfinding algorithms should start simultaneously.	Yes	-
The steps taken and time taken for a pathfinding algorithm should be displayed after it has finished executing.	Partially	The steps taken for a pathfinding algorithm to find the shortest path has not been coded for. This is because it is difficult to define what a ‘step’ is. If we define that a ‘step taken’ is a line executed in assembly level, then the number of steps taken would be directly proportional to the time taken. Thus, I have decided that only the time taken needs to be output

The program should have a ‘step-by-step’ mode, for all instances of the map, where one step of each pathfinding algorithm is done per button press.	No	There was not enough time to attempt this task
The user should be able to resize the grid. If resizing up, then no map data should be lost. If resizing down, then inform the user that data may be lost.	Yes	The user can resize the grid, but there is no message informing the user that data may be lost when resizing down - I considered this to be unnecessary
The user should be able to switch between ‘weighted mode’ and ‘unweighted mode’. In unweighted mode, all current non-empty and non-wall tiles should be treated as a wall, but not deleted off the map.	No	This objective was created due to the BFS algorithm only working for unweighted graphs. However, I have implemented a way that allows the algorithm to treat all tiles, except for “Empty” tiles, as impassable. Therefore, there is no need for an unweighted mode
The user should be able to save the current map file into a text file. The program should also be able to load custom maps.	Partially	The user is able to export the map as a bitmap image and import bitmap images. However the user cannot import/export project files, which includes both map data and tileType data

User Feedback

I have obtained feedback on the program from two end-users, UserA and UserB. UserA had used my pathfinding program multiple times whilst it was in development. UserB has never used the program before. I have given them a set of questions to answer. Their responses are recorded below:

How easy is it to use the program?

UserA: Very easy, however I would want a general explanation on what the difference between the types of searching algorithm. I would also want the program to classify what the tiles do, and if weight affects the algorithm at all.

UserB: Some on-screen help would be useful to provide a workflow for the set-up. Also an on-screen overview of the exploration topic would be useful. Maybe under a Help menu. There were some difficulties due to anomalies in the user interface. Buttons in the Tile Editor were only partially visible. Some were not readable.

How well does the program work as a tile map editor?

UserA: Very easy and simple to use. It is quick and reliable, importing maps as a feature is probably the best part of the tile editor.

UserB: Although I could create new tiles, I could not get them to work. The numerical parameters have no guidance as to their meaning. For example, -1 for Impassable did not seem to make sense when all other tiles were a positive number. For simply black and white tiles, it seemed to work fine.

How well does the program work as a pathfinding comparison tool?

UserA: Very simple and easy to compare each algorithm. Very fast as well. However, I would like to control the speed at which the program solves the path, as I would want to know at each point why it was doing what it was doing.

UserB: The TIME taken to complete the pathfinding for each algorithm was clearly shown - (although, if this is a key feature, maybe emboldening this line would be useful. I did not see other methods of comparison, (e.g. the number of operations carried out) and the information for early runs soon disappears off the top of the feedback panel, with no way of scrolling back up to view it.

Did I learn much about the pathfinding algorithms? Not really, other than how fast they were. I liked the shading on the grid, but it needed more information.

Are there any issues with the program?

UserA: When a path is found, the original map is drawn over by the area it had searched, I would like to be able to see the original map so I can see the exact path it took, maybe put a transparent colour for searched area, so we can see the original map.

UserB: User-defined tiles do not work;
Interface has buttons that are partially obscured and unreadable;
Insufficient user feedback and information about use;

Are there any improvements or additional features you would suggest?

UserA: I would like to control the speed at which the program solves the path, as I would want to know at each point why it was doing what it was doing. Allow the log to scroll so I can see previous log entries. When path is found make the searched area transparent so I can see the map underneath it.

UserB: Splash screen of introduction - to engage students with the topic; Help menu option;
Fix the interface;

Discussion of User Feedback

It is good to hear that UserA found the program easy to use and that it performs well as both a tile map editor and a pathfinding comparison tool. UserB, however, did not seem to find it as easy. This is understandable, as UserB had never used the program before. UserB was likely unaware that BFS works strictly on unweighted graphs, thus the comment about custom tileTypes ‘not working’. As suggested by UserB, a help menu would have definitely been useful for new users.

UserA has mentioned that the original map is drawn over by the search area. This issue could be easily solved by creating a checkbox that toggles the visibility of the search area. This idea could even be extended to toggle the visibility of paths. These options would have to be place in a new ‘Settings’ tab, which could also contain other options that allow the program to be further customised to the user’s liking. This idea is explained in more detail in the ‘Possible Extensions’ section further below.

UserA has also made some suggestions on how to improve the program. Being able to control the speed at which the program finds the shortest path would indeed be useful for analysing how the pathfinding algorithms work. This can be done by first letting a background worker thread handle the expensive pathfinding process. This will allow the rest of the program to remain responsive to user inputs. Then a slider would be created to allow the user to control the pathfinding speed. This could be controlled by making the background worker thread sleep for some time, based on the slider’s position.

UserA also suggests that the log should be able to be scrolled. However, this is something that cannot be done so easily. The reason for disallowing it is due to the listBox log being disabled, so scrolling could not have been done in the first place. The listBox has to be disabled to prevent the user from being able to “select” a line. So allowing scrolling requires having to edit the way the listBox works, or perhaps creating a custom control to display the log.

UserB found that the buttons in the tabs were partially visible and the text being unreadable. After some testing, I found that screen resolutions below 1600 x 900 cause this issue. I was unaware of this as I had only tested the program in the screen resolutions 1600 x 900 and 1920 x 1200. According to rapidtables.com, w3schools.com, and hobo-web.co.uk, the most common screen resolution is 1366 x 768. Redesigning the UI to ensure that the program works on this resolution should fix the problem for most users. However, a better and more permanent approach to solving the problem would be to keep tabCtrl’s width at a constant value, rather than having its value percentage based, so that tabCtrl will be fully visible to the vast majority of users.

UserB mentions that a splash screen could be used. However, as splash screens are used to notify the user that the program is loading, and there is a negligible wait time between the user requesting to start the program and the program being loaded, this splash screen would not be easily visible. During the importing of bitmaps, there may be a long wait time, so a splash screen could be used here, so this is an idea to take into consideration.

Possible Extensions

Given enough time, I would have added more features to the program. For example, the program could be extended to allow pathfinding between more than two nodes. There could be two resulting graphs if this is done - A complete graph, or a minimum spanning tree, which would require another round of pathfinding using the weights of each edge that connects the path points together. Interpolation could be done to the mouse movement so that when adding tileTypes by dragging the mouse with the left button held across the map, a complete line of tileTypes is added rather than fragments of it. A new pathfinding mode could be added where there is only one path point - the source - and the program calculates the shortest path from that point to the tile the mouse is hovering above. The path would be updated as the mouse moves across the map. Finally, for selecting tileTypes, I would store an additional selected tileType for the right mouse button. This is so the user can delete a tileType with a non-empty ‘background colour’ with the right mouse button.

Whilst programming, some bits of code were rushed. In particular, I felt that the best-fit screen code where the program outputs a message box informing the user of an invalid width or height was quite disorganised. If I were to revisit the code, I would make improvements to the code structure and make it easier to understand. Additionally, I would replace the current implementation of the screen and tileType panels with classes rather than subroutines, which would make the code much easier to follow.

While pathfinding, the program highlights the tiles the algorithm has seen, checked, and so on. However these highlights block the colour of other tiles on the map and the user may only want to see the path. If I were to revisit this program again, I would add two pathfinding output modes - One where the user only sees the shortest path, and another where the user sees the tile highlights and the values associated with the pathfinding (e.g. F-Score, Tentative distance). For the second mode, the user can choose between real-time or step-by-step mode where the user clicks a button to move to the next visible pathfinding step.

To make the program more user-friendly and customisable, I would create an undo/redo option so that if the user accidentally fills their map with an unwanted colour, they can undo that change rather than start over again. An autosave function could also be implemented with an editable frequency, along with backup copies of the project, in case of an unexpected crash. I could also create a ‘Settings’ tab, which will allow the user to edit the ‘constant’ values of the program, such as the maximum number of tileTypes or the height and width of program controls. Finally, I could let the BackgroundWorker thread handle expensive operations, such as a pathfinding algorithm being executed, to allow the user to stop pathfinding in the middle of a pathfinding operation.