Comp 4 Programming Project

Dijkstra's Algorithm Learning Resource

Dylan Morroll
George Spencer
Started on 09.09.13

Contents

Analysis

Introduction

Dijkstra's algorithm is a formula used to find the optimal route, customisable to the user's desires, from one location to another. Usually this is a route from a user's current position to their target destination and Dijkstra's algorithm is mainly used in Satellite Navigation devices. Using at least 3 satellites and finding the distance between that satellite and the satnav allows the satnav to work out whereabouts it is on a global scale, as long as it does have said connection to at least 3 satellites. Once the satnav has worked out where it is and where your target destination is, it then works out the shortest path (usually) between the 2 locations using Dijkstra's algorithm. However, with advancing technology, satnavs also have the ability to update the digitally displayed map with any traffic conditions, one way roads or shutdown roads, meaning that some new satnavs have the ability to divert you past these inconveniences if you wish to find the quickest, most efficient route. Devices like these are becoming more and more useful in today's society as environmental issues are becoming more apparent and environmentally friendly devices are becoming more popular.

Dijkstra's algorithm's ability to be able to take into account the length of the path from one point to another is very useful; for example if there were 2 routes, one of which was 10km and the other 7km, then you could use this distance as the length of each route, however you could also use the average time it takes to take both routes and use that as the length. Or you could use this length factor to take into consideration more complex ideas, such as if you will move slower on one path than the other, or how much it will cost. You can combine these and then use Dijkstra's algorithm to analyse, potentially hundreds or thousands, of different options of different routes from one place to another, to find the most efficient and effective route. Multiplying the different factors together in a way that identifies the most important factor allows you to find which would be most suitable for you. Figure 1.1 includes a table of potential questions.

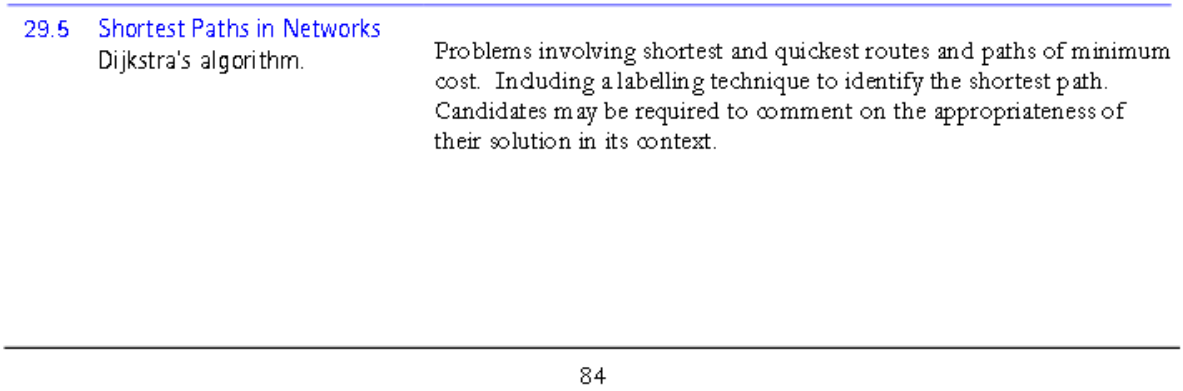
Figure 1.1

Factor	**Route 1**	**Route 2**
Cost	£10	£15
Distance	10km	7km
Speed	20% slower	N/A

Now you can either add them both up in a standard way and use the speed as a multiplier, so route 1 would have a weight of 24 units and route to would be 22 units, so route 2 would be more efficient. However if you value the cost more you could multiply the cost value by 2, in which case route 1 would be 36 units and route 2 would be 37 units, so route 1 would be more efficient. Dijkstra's algorithm has many

practical uses in real life situations as well as being on the specification for AS Further Mathematics, so it is a very useful topic for students to learn, this can be seen in Figure 1.2.

Figure 1.2



29.5 Shortest Paths in Networks
Dijkstra's algorithm.

Problems involving shortest and quickest routes and paths of minimum cost. Including a labelling technique to identify the shortest path. Candidates may be required to comment on the appropriateness of their solution in its context.

84

Source: http://filestore.aqa.org.uk/subjects/AQA-6360-W-SP-13.PDF - 28/10/13

The User

The user of my project will be both students learning A-level further Mathematics and the teacher's teaching it, specifically Mr Simpson. Mr Simpson does not have much experience with ICT and his knowledge of how to work it is very limited. Although he can perform the simple and necessary tasks to work a computer, such as opening any necessary programs for teaching like SIMS (online register program) and opening the internet to access online resources such as MyMaths, beyond that his capability is limited. He doesn't have the skills of being able to use common features throughout different programs to figure out how to use a program, meaning my program will have to be very simple and self-explanatory, possibly including a tutorial, so it can be easily used by not only Mr Simpson, but also any students that wish to use the program in their own time. Although these are the people who I will be designing my program for, those who already have some knowledge of what Dijkstra's algorithm is, it will be available to anyone who takes an interest in the topic and wants to use my program.

The Investigation

To find out about current system of how Dijkstra's algorithm is taught I am planning to do a few things. Since I am not currently studying further mathematics I am not sure of how it is taught, however I do know some people in my college who are studying it. I am planning to arrange a meeting with a teacher, probably Mr Simpson, on how he currently teaches Dijkstra's algorithm to his students and the resources he has available to aid in his teaching. Furthermore I then plan to ask any students that have studied Dijkstra's algorithm how their experience was when learning about it in lesson, how useful any available resources were and also how

interesting it was to learn about it, this will help me evaluate the usefulness of my project. Since it was taught in the first year of further mathematics I will also be able to inquire about what it was like revising the topic later on throughout the year and the resources they had available to them to do so outside of the lesson. Once I know these things I will be able to customise my system more to accompany for anything the students feel like they are currently lacking either in lesson or out of lesson.

A Description of the Current System

The current method of teaching Dijkstra's algorithm is like most standard Maths lessons. Firstly Mr Simpson runs through a presentation (usually a PowerPoint made by the Maths department, but sometimes a MyMaths resource) about what the algorithm is and the task that it performs (in this case finding the optimal route between two destinations) and just a few key ideas about it, such as its applications in real life in satellite navigation technology. Sir then runs through some easy examples and gets the students to try and figure out the shortest route, then moves onto a much larger diagram with many more routes and the students quickly realise that it would take a ridiculous amount of time to compare the possible dozens of different routes, which is where he introduces Dijkstra's algorithm, explaining that it would make situations like this much easier and quicker to resolve. After introducing the students to the formula, sir continues by running through an example question using the algorithm, the students continue by first applying it to much easier, smaller, examples where they can get to grips with how solve questions which the algorithm and how to apply it. He then progressively moves the students onto harder example questions, possibly involving questions that are a little different to others, such as a situation where two routes have the same length and waits for the students to spot it and asks them what they think they should do before actually explaining the answer. I think this is a good teaching method employed by Mr Simpson since it makes the students think for themselves so they're more likely to remember what they have learnt and also teaches students one of the most important skills of thinking for themselves. After this introduction to the topic, taking usually 20-30 minutes including the example questions, sir then sets an exercise out of the text book for the students to do for the rest of the lesson, where they check their answers against those in the back of the book and ask Mr Simpson if there's anything they are struggling with or don't understand, then the rest of the exercise is usually set as independent study to do at home

How to Solve a Dijkstra's Algorithm Question

Here I will show how to work through a Dijkstra's algorithm question step by step. For this question I will be working through the example question used in the learning resource MyMaths, which Mr Simpson uses when teaching the topic. When first approaching the topic it gives you the steps used to solve the question, so you can apply them to the example question.

Dijkstra's Algorithm: Labelling the Vertices

Step 1: Give the start vertex (S) a permanent label of 0.

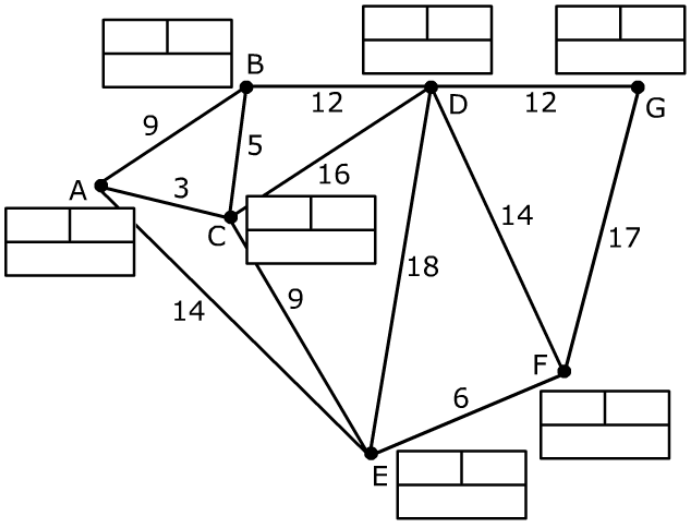
Step 2: Give each vertex connected to S a working value by recording its distance from S.

Step 3: Find the smallest working value and give the corresponding vertex V this value as a permanent label.

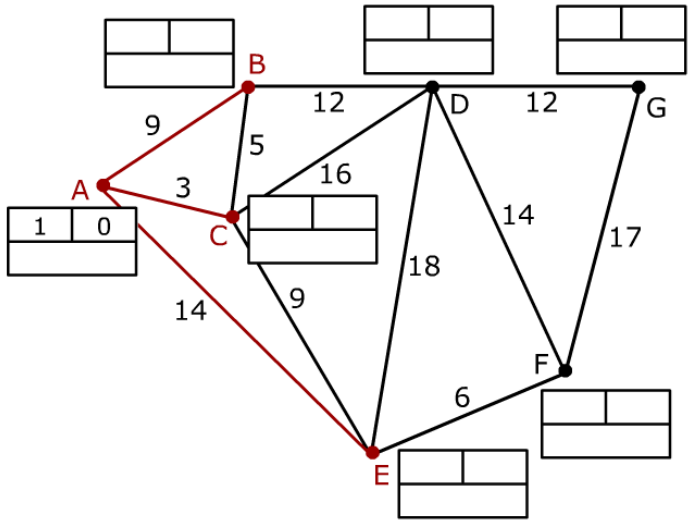
Step 4: Update working values at any unlabelled vertices that can be reached from V.

Step 5: Repeat Steps 3 and 4 until the destination vertex has been given a permanent label.

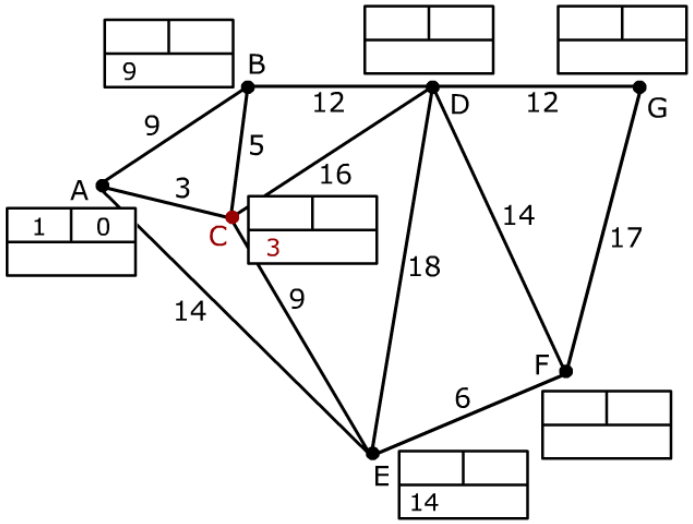
Here we are given an example grid layout which you are prompted to work through step by step and then compare your answers to the step by step example MyMaths provides.



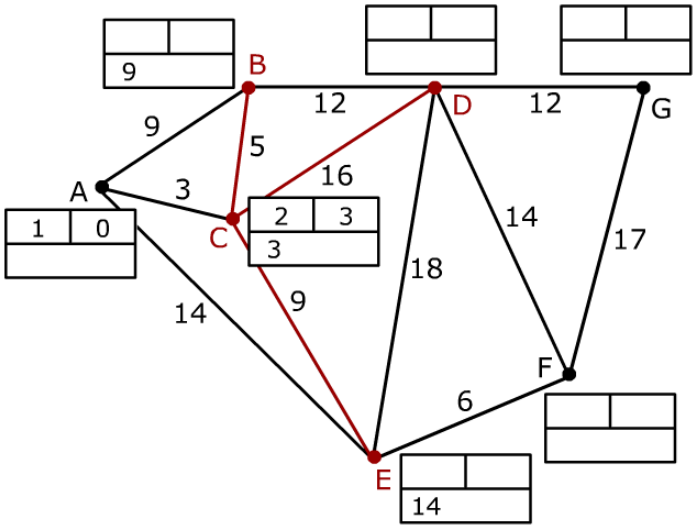
To start we need to label the start vertex and give it a permanent label of 0, then look at any branches leading of it.



The vertexes that are connected to our start vertex are highlighted; we give each vertex a working value of the distance between it and the start vertex and find the shortest distance.

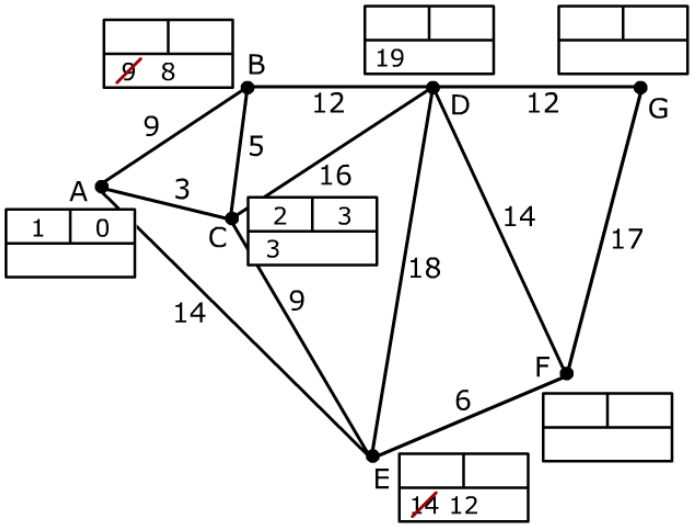


Here the shortest distance is to C, so the working value is turned to a permanent value and it is given the next permanent label, this time 2.

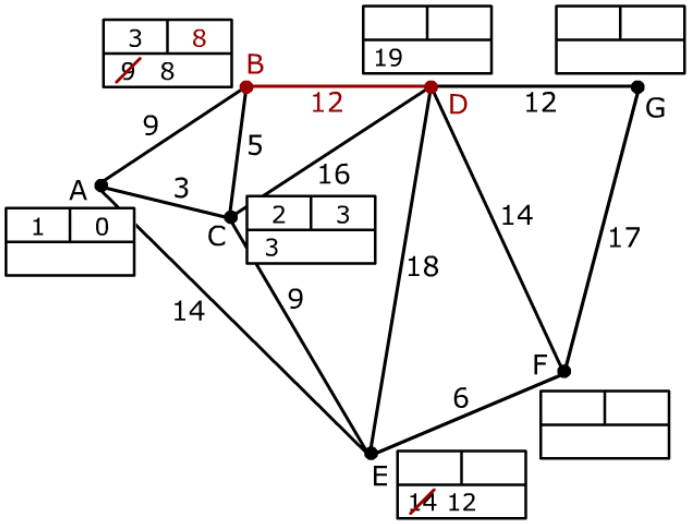


Now we have our next permanent value, we look at any vertexes that are connected to our permanent value and update of the any working values for the

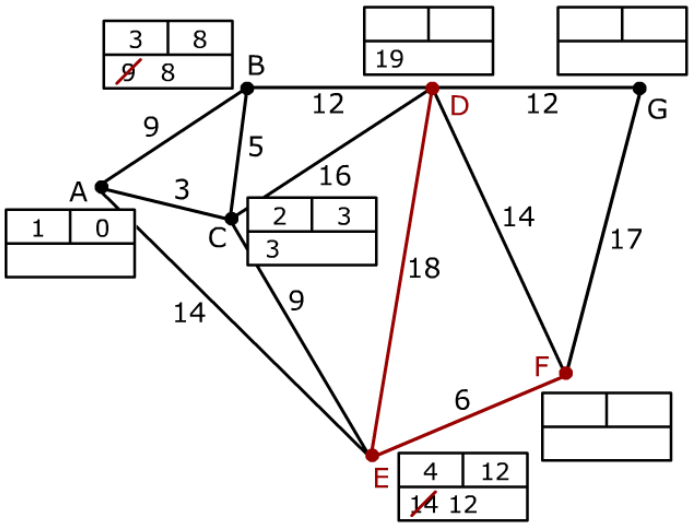
vertexes that can be accessed. To find our working value this time we add the permanent value of the vertex we have come from and add it to the distance to our next vertex.



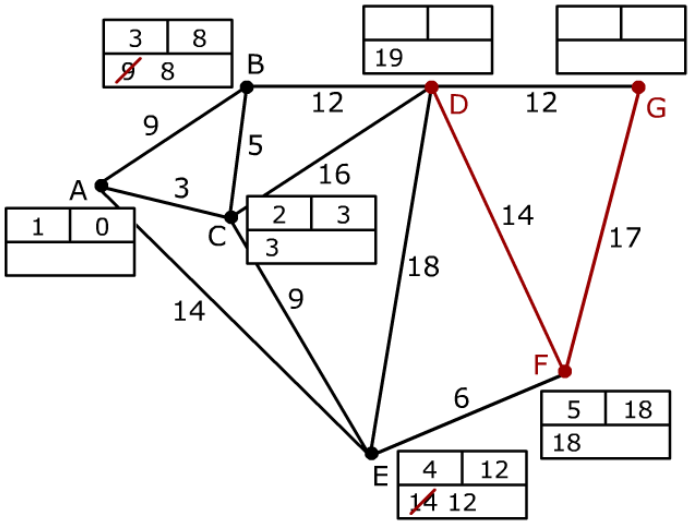
We add a working value to D since it didn't have one and for B and E we change it since the working distance this time is now shorter (D: 3+5 < 9, E: 3+9 < 14).

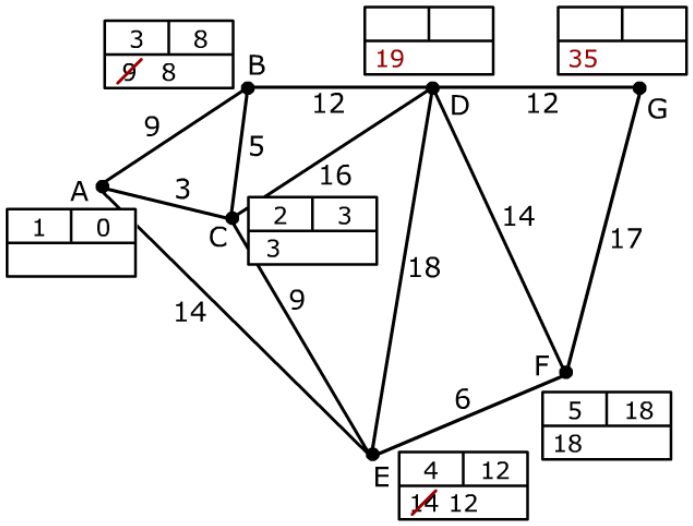


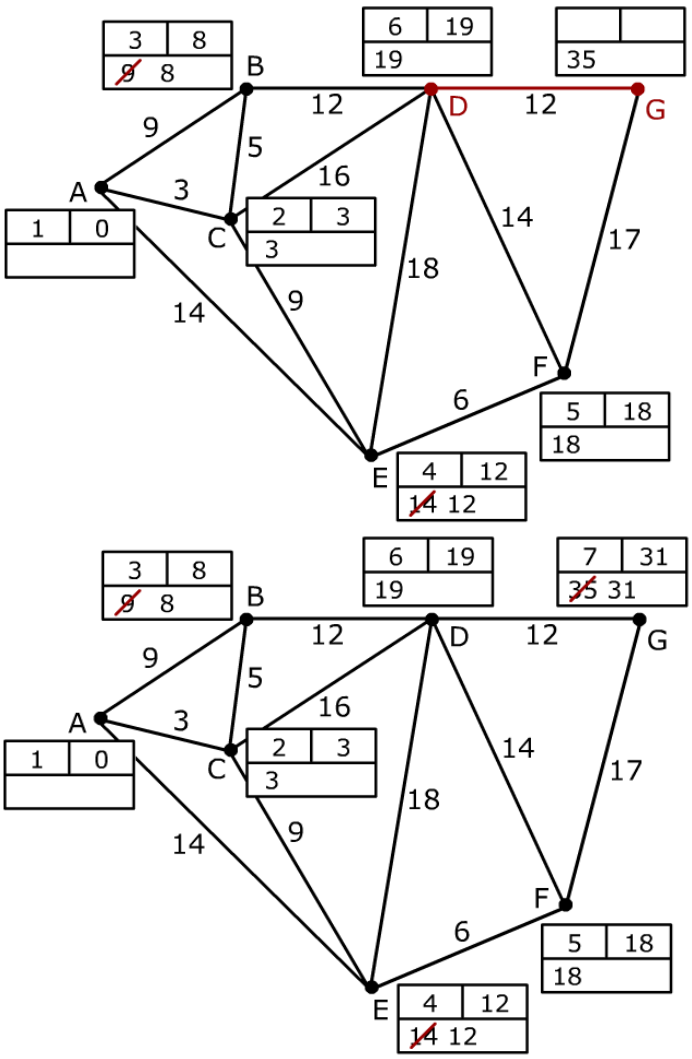
We then look at our next shortest value, which is a B with a value of 8, update it to a permanent value and give it a permanent label of 3. We then look at any vertexes that can be accessed from hear which we haven't already been to, which is only D. Since the distance to D from B would be 20 (8+12) we don't change the working value since it is shorter than our current value for D.



Since the working value for D was shorter than what we already had we look at any working values which don't have a permanent value, take the shortest one and work from there, in this case E.







The destination vertex has now been labelled so we can stop. Now we have to do the reverse process to find the shortest route.

Dijkstra's Algorithm: Backtracking

Step 1: Start from the destination vertex.

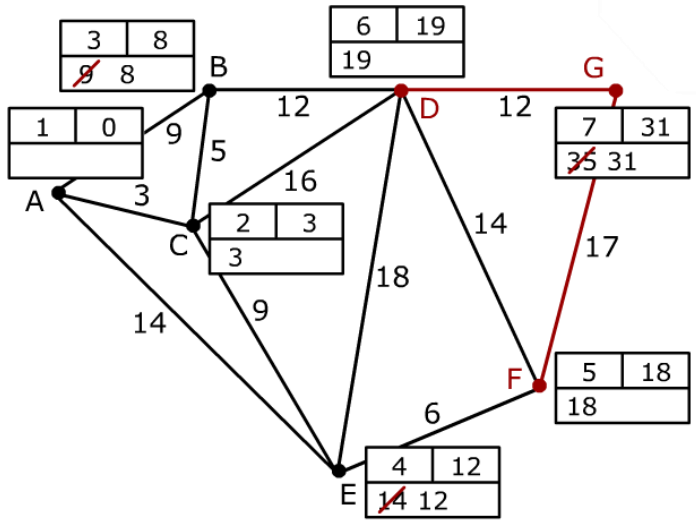
Step 2: From the current vertex V, look at all the vertices that lead directly to V.

Step 3: Of these vertices, vertex P is the previous vertex on the route if:

$$\text{label at V} - \text{label at P} = \text{weight PV}$$

Step 4: Repeat Steps 2 and 3 using the vertex just found as the current vertex.

Step 5: Stop when you have backtracked all the way through the graph to the start vertex.



Now we look at the 2 vertexes accessible from our destination point, G, these are D and F. To access F; 31 - 18 = 13, whereas our marked down value is 17, so our route can't include F. To get to D; 31 - 19 = 12, this is the same as our value for this vertex so D must be included in the route. You just repeat this process until you get to the start point, so from D you go to C, since 19 - 3 = 16 and from C you go to A, since 3 - 0 = 3. So the shortest route from start to destination vertex is: A, C, D then G.

Data Dictionary of the Current System

Name	Data Type	Length	Validation	Example Data	Comment
Vertex Y	String	12	Check that it is in the	(7, 3)	Provides a source of

	(x, y)		format X*Y and that both x and y are positive integers.		identification for all the vertexes apart from the starting and ending position.
Current Vertex	String	12	Check that the current vertex matches up to one of the named vertexes.	Vertex 7	Holds the name of the current vertex
Current Vertex Weight	Integer	3	Check that it is a positive integer.	12	Holds the current value for the distance to the next block from the current block/ weight of vertex.
Starting Position	String (x, y)	12	Check that it is in the format X*Y and that both x and y are positive integers.	(2, 2)	Holds the co-ordinates for the starting position.
End Position	String (x, y)	12	Check that it is in the format X*Y and that both x and y are positive integers.	(7, 9)	Holds the co-ordinates for the ending position.
Temporary Value Y	Integer	3	Check that it is a positive integer.	5	Holds any temporary values currently assigned to vertex Y
Permanent Value Y	Integer	3	Check that it is a positive integer.	8	Holds the permanent value for vertex Y

Limitations of the Current System

After doing research into the current system to find out how it was taught, I also took the opportunity to look at some limitations of the current system and have found out that the way that Dijkstra's algorithm isn't very exciting, to say the least. One student said "even though I enjoy maths, I did find this topic could've used something to make it a bit more interesting." Although some may disagree, on a whole Dijkstra's algorithm is a very dull topic, not because the algorithm itself, or learning about it, is dull, but because it takes so long to work through a single question. You have to run through a lot of possible options and notate the current length of the route you are working on and compare it to other routes etc. then work backwards to actually find the shortest route. This means that it can become very boring to do question after question and after the student loses enthusiasm when revising a topic they often struggle to get the willpower to return to the topic, or any revision for that matter.

There aren't many resources available either, to both the teacher and the students, Mr Simpson even commented that "apart from 'MyMaths' (an online resource) a handful of old whiteboards and textbooks, there's not really many resources. We have what we need to teach the topic, but nothing to make it exciting and I often find students struggle with the topic in the examination as they get bored easily when trying to revise the topic". There are a few different textbooks

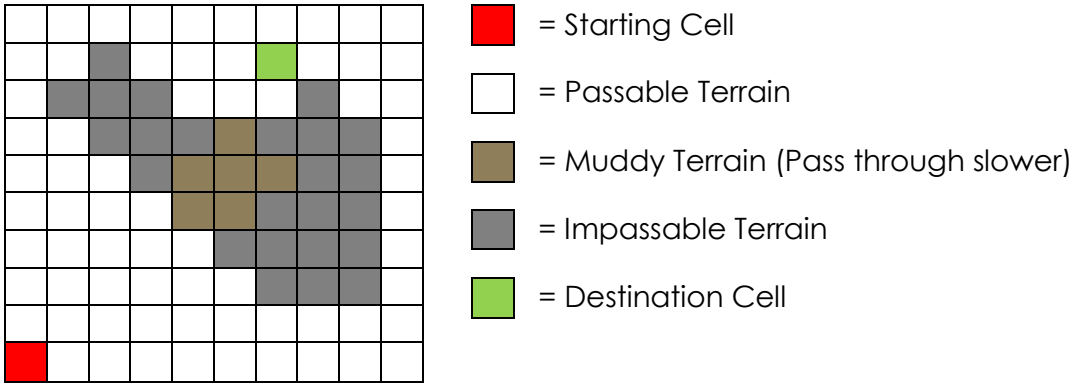
that can be used to get example questions from and a few different diagrams that help explain the example, but again these questions can become very boring and repetitive for the student to work through, especially since it takes so long to draw out the diagram and add the notations to it. There are also a few online resources available; however these are usually for the introduction of the topic only offering a couple of questions that are the same each time you run through the presentation.

Data Flow Diagram for Current System

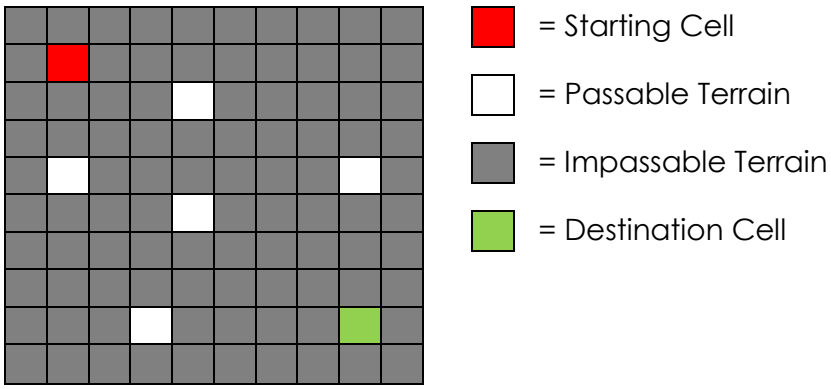
Data Flow Diagram for the Prosed System

Description of the New System

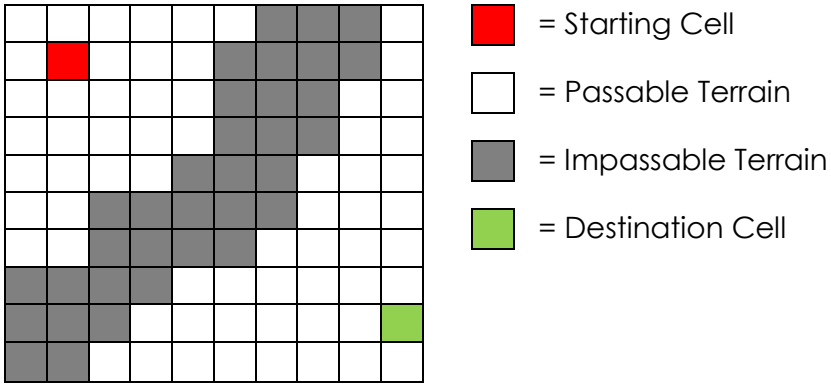
For my project I am aiming to create a visual learning aid that demonstrates how Dijkstra's algorithm works. Dijkstra's algorithm is an algorithm that solves a very visual problem, finding the shortest distance between two positions, but it means that visual representations of how the algorithm works are very beneficial to helping students understand the topic. I am planning to have a grid of varying sizes which can be changed by the user for optimal use, then to decide on which of these blocks will be in use, and the weight of the blocks in between to simulate changes in terrain etc. For example you could have a 10x10 grid, such as the grids below, and once you have assigned attributes to the cells, you can then get it to work out the shortest route.



This element of my program will be more for demonstrating how Dijkstra's algorithm works. The other element of my program can be seen below; using the exact same method as above of assigning values to each cell you can create a more complicated Dijkstra's algorithm problem by only leaving a few cells open for use, my program will then use Pythagoras Theorem (rounded to nearest whole number) to find the distance between available cells for use, or allow the user to input a value, to then perform a more standard Dijkstra's algorithm question.



You can also use a combination of these two features, however due to the fact that any areas where it has to pass over impassable terrain it will have to work out the shortest path from every block on this side connected to the impassable terrain, any working out (working values, permanent values etc.) cannot be displayed as there will be too much in too small of an area.



Objectives for the New System

I think my project will prove to be a very useful teaching tool as it will have two purposes, during the introduction of the topic the first function of my project will provide a good demonstration of just what Dijkstra's algorithm is and a visual representation of a different use for the algorithm than the standard one everyone thinks of with its use in satnavs. Then after Mr Simpson has been through some of the fundamental aspects of the algorithm, such as the formula, what it's used for, how to work out questions that require Dijkstra's algorithm and examples, the second part of my project will allow for customised examples to be given to the students to work out, then my program will work out the shortest path so that the answer can be compared with what the students got.

I feel like the first part of my project that shows a demonstration of what Dijkstra's algorithm is and what it does is very unique as many students said they never had anything similar to that as a learning aid when studying the topic. The only other available resource is the whiteboards that are available for use in every subject, although they do allow the teacher to give the students something different to do, where they can all interact with the teacher and one another, it doesn't provide many possibilities. Its uses are limited to quick sketches and possibly a small multiple choice game or something similar, usually used at the start or end of teaching the topic to see what the students already know about the topic or to recap what they have learnt.

Make it less boring - good ideas with sir letting us run through an example w/o Dijkstra's - it is maths (lots of practice Qs) - however boring to do questions - aren't challenging

Constraints and Limitations

One of the limitations of my project is the technology that can be used to access my program. For Mr Simpson to be able to use my program in his lessons as a learning tool he will have to access to both a computer and a functioning projector, although every Maths class has access to these items of equipment, with my college also functioning as a school for over a thousand pupils, technology often gets damaged or classes get moved to different locations, classrooms which possibly

don't have access to this technology. If either of these possibilities becomes a reality in the lesson Mr Simpson was planning to teach the students Dijkstra's algorithm, it will mean sir won't have access to my program for its use. Expanding on the idea of technology is the lack of access to computers for the students in Maths lessons, there is only one Maths room out of about 10 which uses computers and this classroom is usually occupied by pupils in the school and college students are rarely, if ever, in this classroom. This means that the students won't have access to my project unless they can get access to a computer in their own time which might be troublesome if they don't have one at home, or they aren't allowed to use computers due to religious issues. Furthermore software is an issue as well, some of the programs/languages I am considering for creating my program in aren't compatible with older versions of Windows or different operating systems, or they require potentially expensive pieces of software meaning that I will have to try and create a very compatible program so (hopefully) all the students can access it at school and at home.

Another limitation, which I have already partially covered, is the limited ICT capabilities of some students and teachers. Although students being taught now, in the information age, are much more familiar and comfortable with technology than perhaps those that were taught a few a decades ago, there are still some students whose ability to use said technology is very limited. Especially since the program I am creating is for use in a Maths lesson which doesn't require advanced capabilities in the use of technology. This means that my program will have to fairly simple and self-explanatory, so that even if Mr Simpson knows how to use it through practice, students who wish to use it in their own time can do so with ease.

File size, saving (?) files on 250Mb space etc. - sir email examples

Consideration of Possible Solutions

Flash Based Solution

Flash is a program that is great for creating animations that can be interacted with, it is often used for animations on websites and has great uses for all users. It has the potential to easily create basic animations if you are just getting to grips with flash, to being able to create very professional looking animations as well. The online resource MyMaths uses a lot of flash animations to give it a warm, professional, welcome. Using a program like flash would allow me to easily create a good, complex, visual representation of Dijkstra's algorithm, whilst also allowing the users being able to interact with it. However I feel like the program I am creating does not need a very complex visual display and would require more in the area of complex programming code to be able to work out a problem using Dijkstra's algorithm, which isn't easily accomplishable in ActionScript (flash's programming language).

Objective	Suitability
Create a grid of	ActionScript would be a suitable language to accomplish this in.

varying sizes	
Give a value/weight of each cell	ActionScript's heavily visual focused programming style would easily allow for something like this.
Calculate distance between 2 cells	A simple calculation like this should be easily achievable for any programming language, including ActionScript.
Allow for user inputted values	Flash allows for a lot of interaction in its programs, so it would be very easy to have a visual input box on screen.
Randomly generate a graph	Although it might not be as efficient as other programming languages it could be done in ActionScript.
Work out a problem using Dijkstra's algorithm	Similar to the previous objective, this could be achieved in ActionScript, however it might be much more difficult to do when compared with other programs and might take up a lot more code.
System is easy to use	Flash's heavy focus on the visual aspect of a program would means it would be easy to create a very simple and easy to understand GUI for my program.

Python Based Solution

Python is a program that, quite differently to flash, focuses more on the script of the program rather than the visual aspect of it. To create a GUI in Python you have to import a separate function into Python before you can begin to program a visual interface and what can be created doesn't match up to other languages, such as ActionScript, which focus on the visual interface of a program more so. However, Python allows the users to program in a variety of different ways, which is very useful as you can switch between them during a project to find the most useful for each part of the project, these are:

- Imperative: Where the code is executed line by line in the order it was written
- Functional: Where the program calls a series of functions (usually mathematical) to be executed, this type of programming uses no variables.
- Object Oriented: Where the code is written in chunks, called functions or classes and the user can call these at any time in the code to reuse part of the code and make it more efficient.
- Event Driven: Where part of the code is executed when an event happens, such as a button being clicked in the GUI.

Objective	Suitability
Create a grid of varying sizes	Python can create grids very easily both its Tkinter and Python Game GUI packages; however they might not meet the aesthetical standard of a program such as flash.
Give a value/weight of each cell	Python's easy ability to create variables and lists means it would be easy to assign a value to each cell.
Calculate distance	This simple mathematical task would be easy to perform in Python.

between 2 cells	
Allow for user inputted values	Assigning inputs to variables using the "input" and "raw_input" functions means it is very easy to get user inputted values.
Randomly generate a graph	Python's "random" package that can be imported gives a lot of different ways to create random integers or strings which can be easily used to create a graph.
Work out a problem using Dijkstra's algorithm	Python's focus on the script of the program and versatility means this would be more easily accomplished than in the other programming languages I have mentioned.
System is easy to use	Although Python's GUI packages might not be too easy to understand, I feel like with enough feedback on my designs the efficiency in which you can program in Python, I could limit the amount of clutter that will be on screen to simplify the GUI.

Justification of Chosen Solution

In the end I have decided to choose Python as the programming language that I will use in creating my project. ActionScript has some great features, especially its fantastic visual displays and animations that can be easily created in this language when you compare it to some of the other programming language. It will be hard to create a GUI that matches the standards of what I could create in flash, I feel like its heavy focus on the visual aspect of programming lets it down when you try and program a complicated algorithm, such as Dijkstra's. Python's ability to be allow the user to code in so many different ways: imperative; functional; object oriented and event-driven allows so much versatility when carrying out a large project such as this. Although I could program Dijkstra's in ActionScript, I feel that it would be much more efficient, effective and easier to program in Python, especially considering that I have much more experience in Python than in ActionScript.

Design

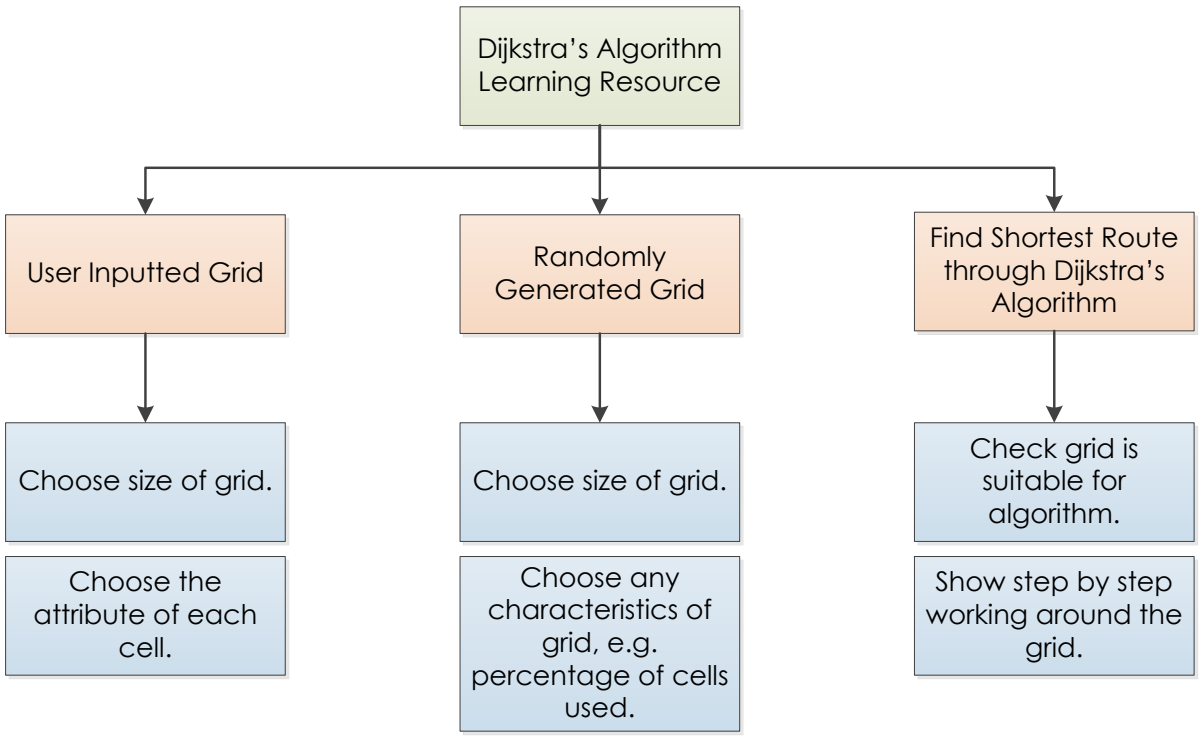
Overall System Design

IPSO Chart

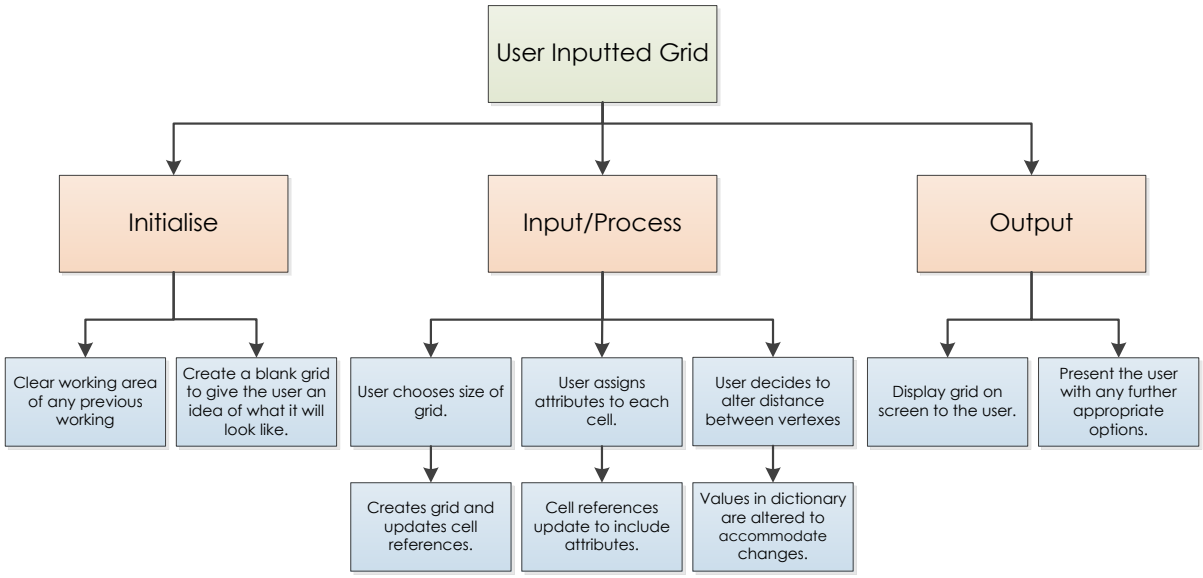
Input	Process	Storage	Output
User chooses the size of the grid.	The grid of size defined by the user is created along with a dictionary for grid references.	Size of graph is saved as an object and the foundations for grid references to be created are stored.	The grid is displayed on screen and further options are displayed.
User decides to create the graph themselves.	Variables created within dictionary when attribute of a cell is changed.	Variables are created and stored within the dictionary.	Visual representation of said attribute is displayed on the GUI for the corresponding cell of the grid.
User selects the cells they want to use and any attributes.	Values are updated to correspond to what the user wants.	The value of each cell will be stored alongside their cell reference.	The cells will be changed in the system and on the user interface.
User decides to randomly generate a grid, either to find the shortest route or display working.	Attributes for cell references are randomly generated. They can then be customised by the user.	Attributes are added to any altered cell references in the dictionary.	The new grid is displayed on screen showing what attributes have been generated for each cell.
User changes any values of distance worked out by using Pythagoras theorem.	System recognises which distance is being edited and creates a variable for the new value.	The value will be stored as a global variable for reference in other modules throughout the rest of the program.	The new value will be visually displayed on the graph.
User decides to solve graph with Dijkstra's algorithm	If there multiple attributes, show the shortest route. Otherwise show how to solve Dijkstra's algorithm.	Solution will be stored in a temporary format until the user deletes it or exits the program.	The, now completed, Dijkstra's algorithm problem and its solution will be displayed on screen.

Modular Structure of the System

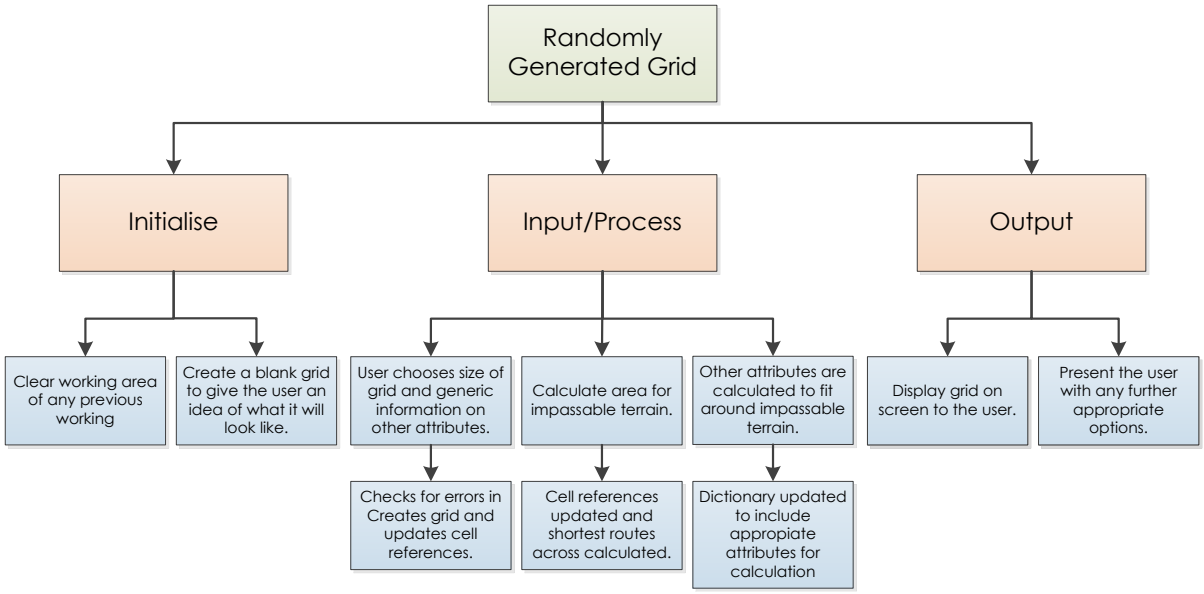
1.



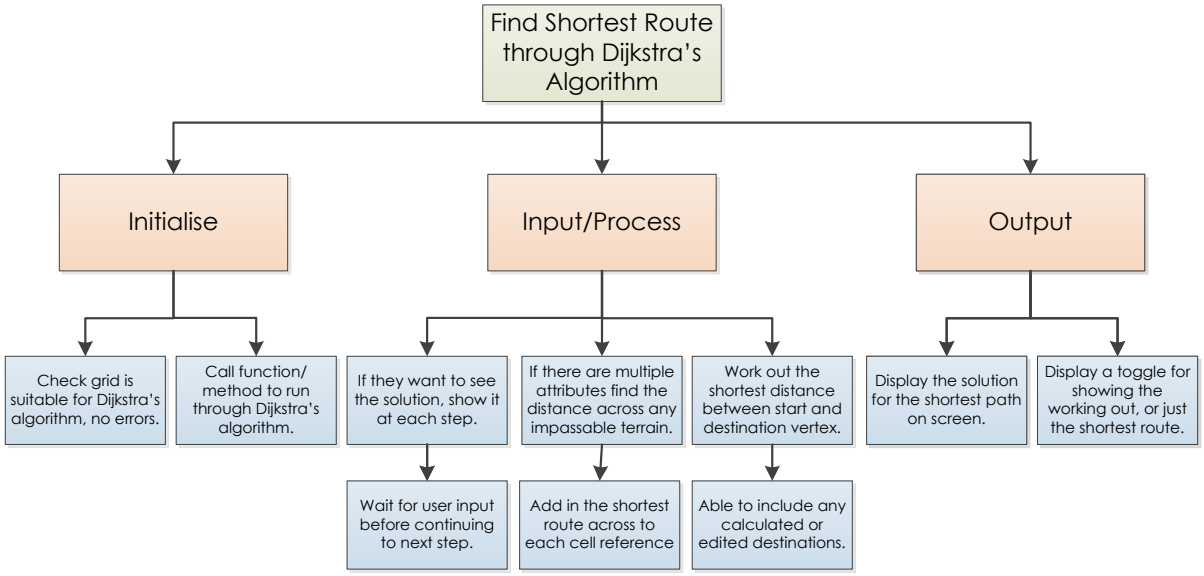
2.



3.



4.



Design Data Dictionary

Name	Data Type	Length	Validation	Example Data	Comment
cells_dictionary	Dictionary	0 to 10000 (longest length of dictionary)	Check that cell ids are within the grid size and are positive. Check attributes are within those specified.	{[0,6]:4, [0,7]:2]	This dictionary will contain a set of co-ordinates for each cell along with their attribute, it will automatically update with editing of the grid.
cell_id	Integer	0 to 3	Check that it is a positive integer within the length of the	57	This will contain the position of a cell within the

			dictionary.		dictionary.
size_of_grid	String(x,y)	0 to 7	Make sure both values are positive integers less than 100.	(57,34)	This will hold the size of the grid, allowing the dictionary to be updated with cell references.
temperary_value	Integer	0 to 10	Make sure it is a positive integer.	5	Holds the current temporary value in use.
working_values	List	0 to 50	Doesn't require validation.	[17, 14, 12, 8]	A list of any current working values assigned to a cell.
permanent_values	Dictionary	0 to 30	Make sure permanent values are sequential and the cell id it is allocated to exists.	{1:4, 2:24, 3:25)	Stores the permanent value of a list along with the cell id.
display_method	Boolean	N/A	Make sure the value is either True or False.	True/ False	Holds the value of whether or not the method should be displayed.

Storage Media and Format

I will be storing my program on the local network that is at my college. Since it is a LAN the G: drive at my college is available for access on all computers by both teachers and students, this means that I can store it on this drive in an easily accessible area so everyone can access it. I will be saving my program as an .py file, since my college runs on a shared network, an application or program only needs to be installed once and every computer can use it, due to this when my college installed Python for the use of computing students, every computer has access to it meaning that every computer can open files with the .py extension. Python files are easily accessible, as you can simply double click them and run them through command prompt, since my program will be using a GUI to interact with the user, the command prompt will simply run in the background and not affect the user. By using a the python extension it means I not only save space on the shared network by getting rid of the need for another program to access it, but also it means that any students who want to can edit it in Pyscripter and read through and edit the code.

Identification of Processes and Suitable Algorithms for Data Transformation

User Creating a Grid

When the user decides to create a grid, they will be prompted with a series of options, which will then update the user interface. Hi

Function create_grid
    Grid_size --> "Enter the size of the grid"
    Write "Click on an attribute on the right, then highlight the cells you wish to apply this attribute to, if working is to be displayed only use passable and impassable terrain and leave at least 5 cells space between each passable terrain cell."

```
        While continue isn't clicked:
                Attribute --> user_inputted attribute
                On click (on area of the grid):
                        Get mouse position
                        Distance_x --> assign x position
                        Distance_y --> assign y position
                        Cell --> (Distance_x, Distance_y)
                        Add cell attribute to cell dictionary
        EndWhile
EndFunction
```

Generating a Random Grid

If the user doesn't want to create their own grid they can choose to randomly generate a grid with attributes using this function, the user has the option to add parameters to the grid first.

```
Function randomly_generate_grid
        Random_grid --> Get user input; display "Do you wish to enter the size of the grid?"
        If Random_grid --> Yes
                Grid_size --> "Enter the size of the grid"
        Write "Enter any further parameters for the grid below, leave blank if necessary"
        Method --> Get user input, display "If you wish the working out to be displayed select this option, it will limit the grid to only passable and impassable terrain and leave at least 5 cells between each passable terrain cell."
        Slow_terrain --> Get user input, display "Select this if you wish to disable the use of slow terrain"
        User_start_position --> Get user input, display "Select this if you wish to choose a customised start position."
        User_end_position --> Get user input, display "Select this if you wish to choose a customised destination position."
        Generate grid
        If User_start_position --> Yes
                Start_position --> Get user input
        If User_end_position --> Yes
                End_position --> Get user input
        If Method --> Yes
                While passable terrain < (size_of_grid / 20)
                        Current_cell --> Random cell
                        If Current_cell within 5 cells of passable terrain
                                Current_cell --> Random cell
                EndWhile
        Elif Method does not --> Yes
                Cells_with_attributes --> 0
                While Cells_with_attributes < (size of grid / 5)
                Current_cell --> Random cell
                        Attribute --> Random attribute
                        Add Attribute to Current_cell
                        Cells_with_attributes + 1
                EndWhile
```

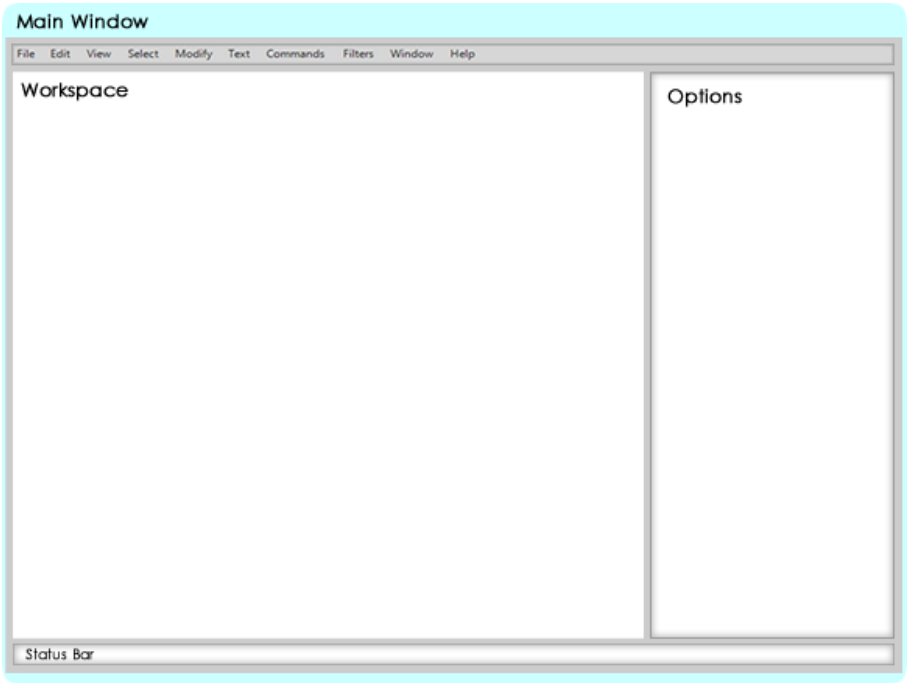
```
EndFunction
```

Solving the Grid using Dijkstra's Algorithm

```
Function dijkstra's_algorithm
Current_cell --> Start_position
While Current_cell does not --> destination vertex:
        Possible_destinations --> [(Current_cell(x) +1, Current_cell(y)), (Current_cell(x),
        Current_cell(y) +1)]
        Current_distance --> working value of Current_cell, plus distance to
destination cell

        If Possible_destinations [1] or Possible_destinations [2] --> Impassable terrain:
                Destination --> Find closest passable terrain on other side
                Distance --> Distance between Destination and Current_cell
                Current_distance --> Distance

        If Current_distance < Possible_destinations [1] working value or
Possible_destinations [1] does not have working value
                Possible_destinations [1] working value --> Current_distance

        If Current_distance < Possible_destinations [2] working value or
Possible_destinations [2] does not have working value
                Possible_destinations [2] working value --> Current_distance

        Lowest_working_value --> Cell with lowest working value
        Current_cell --> Lowest_working_value

EndWhile
```

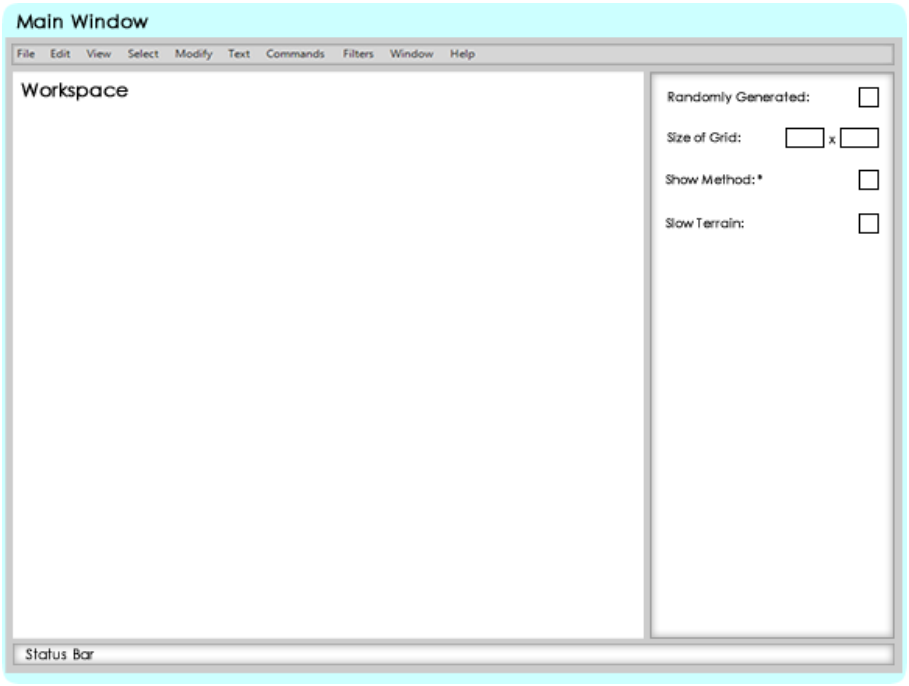
Designs

1.



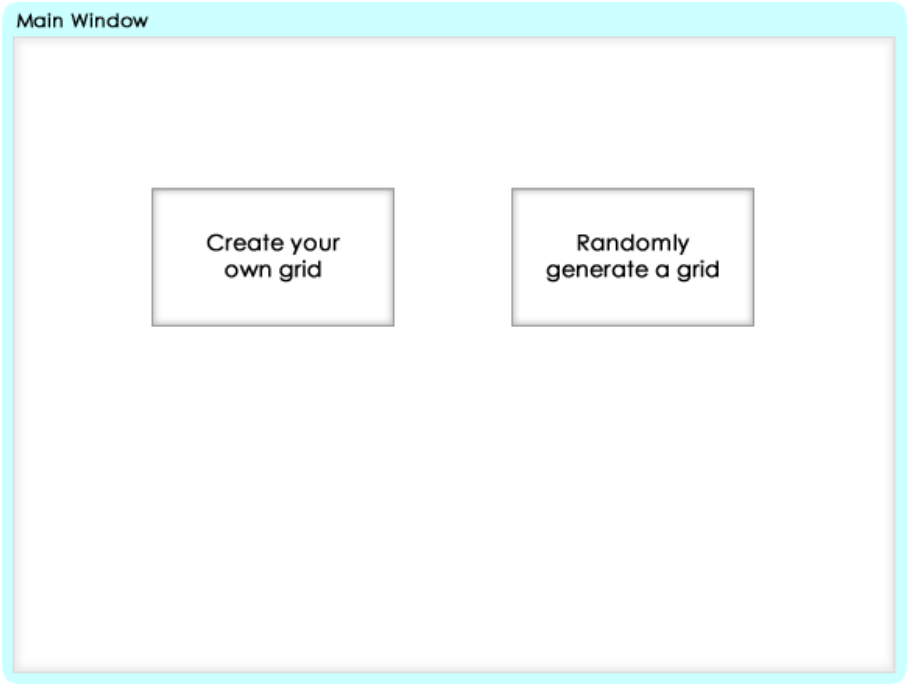
This was the first design I came up with, it was a very basic layout with the options located on the right, the workspace on the left and any options to do with the programs functionality were located on a toolbar at the top.

2.

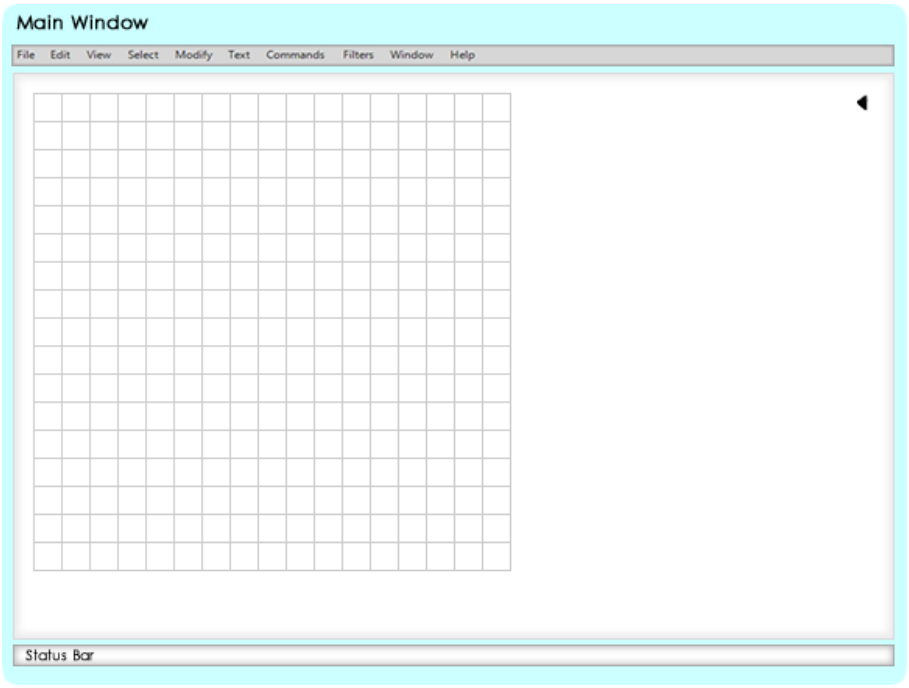


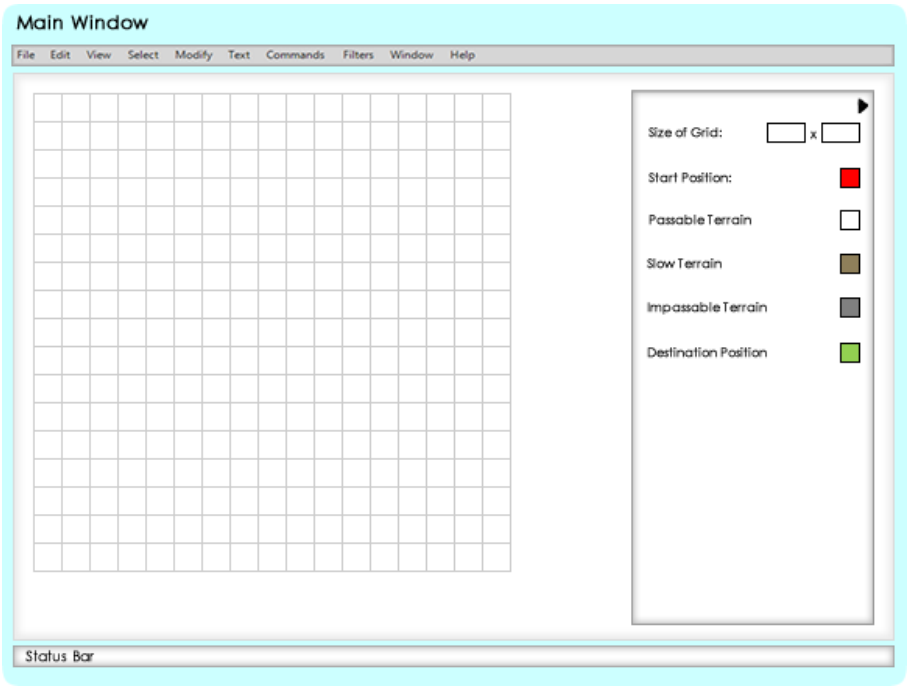
This is the second design; all I did here was to fill in some of the options to give an idea of how it would look with it filled in.

3.



4.





Here was where I refined my idea of the workspace a little bit more, I realised that having an area at the side of the workspace dedicated to the options wasn't very efficient, there was a lot of space wasted at the bottom of the options that wasn't being used and it limited the area of the workspace. Therefore I decided to change my design so that was a little arrow you could click to bring up the options menu and then you could close it when you weren't editing the attributes or options of the grid so that you could get a clearer view of the workspace, especially if the grid was larger than the space available on the screen.

Feedback

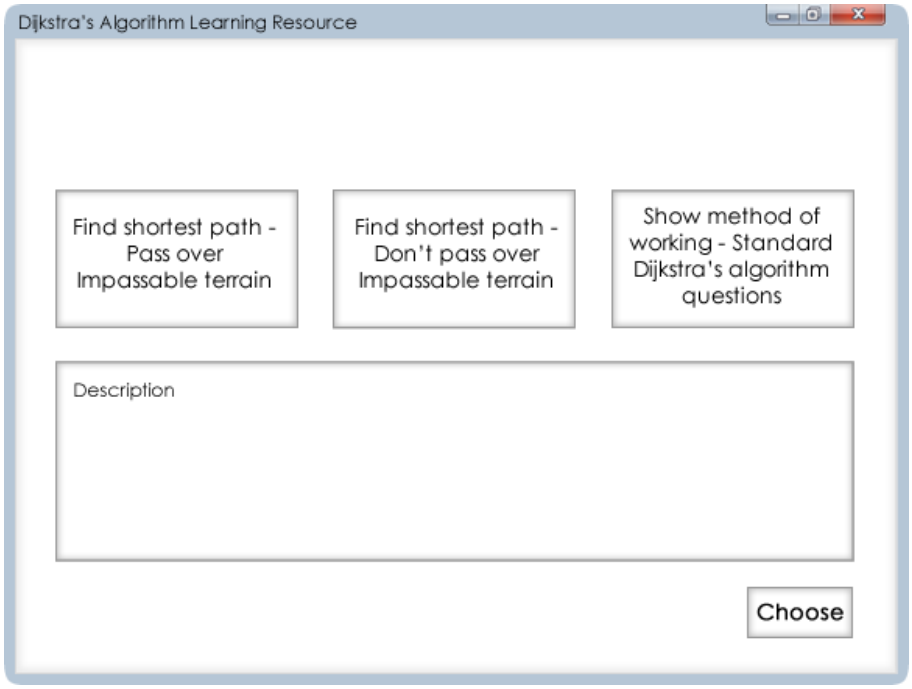
The majority of the feedback I received on my initial designs was very positive. A lot of people preferred the shift from a permanent options area to one that you could open and minimise when you wanted to. A fellow student mentioned "I prefer the direction you have taken with your second design; I feel it is more unique to other programs out there and makes it look more professional and sleek. However there were still improvements to be made, a teacher pointed out that "the functions toolbar and status bar at the top and bottom of your work stick out and make the design look rather clunky." After re-evaluating my designs I had to agree with this statement, I decided to get rid of the status bar and in my program I will be implementing a tutorial, where it will come up with a little help box telling what to do on that screen if you are lost.

Refined Designs

1.

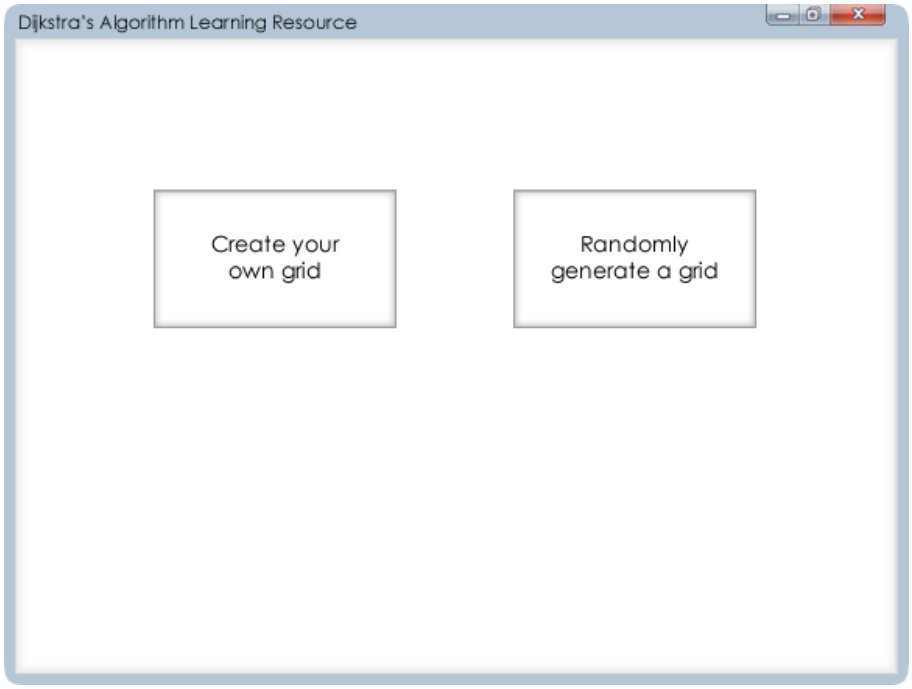
Whilst refining my designs, I also decided to include designs for the menu system as well; this is a very basic idea of what my homepage will look like. You can also see the option to enable or disable the tutorial. I may decide to add to the homepage when I create the actual program if I have any ideas that would fill the empty space whilst still retaining its professional look. I also decided to add the minimize and close buttons to it, change the colour around the outside of the program and change the name, just to give a feel of what the finalised program will look like.

2.



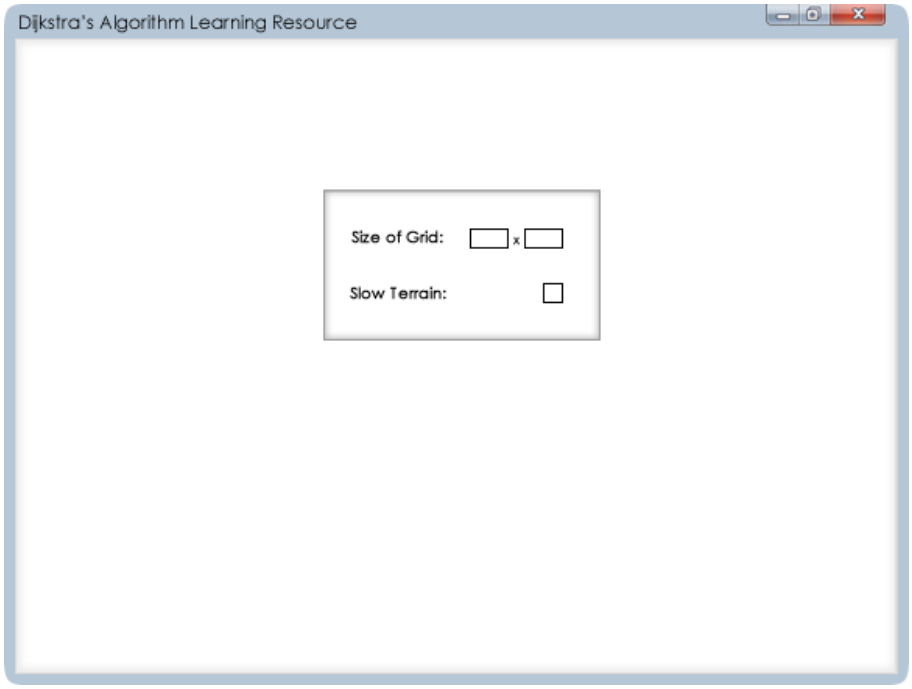
This is the first set of options you will see after clicking start and this is an idea I came up with whilst editing one of the menu pages. Basically instead of having to try and edit your grid to fit certain parameters to get options like the method to show, you will click one of these three options and the parameters will be enforced. The first two options are both the first function of my program, where you will include different types of terrain and find the shortest distance. Option one it will go around impassable terrain and the second option will make the program perform Pythagoras to find the distances over impassable terrain rounded up to the nearest integer. The last option will be covered more in designs 8-11. A description of each option will appear under it so the user fully understands what they are choosing.

3.



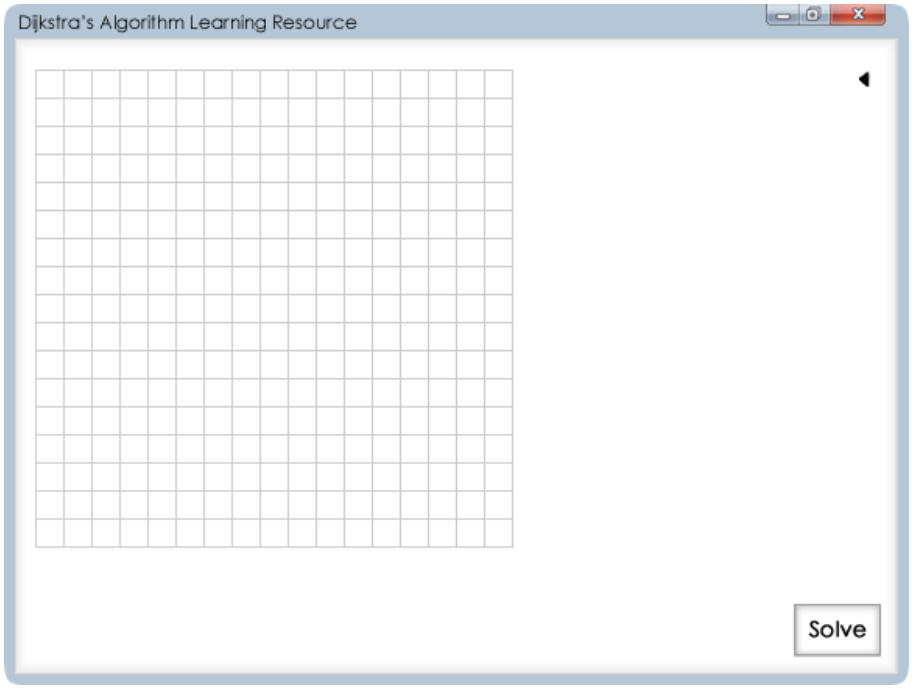
After selecting what functionality of the program you would like to use, you then decide whether or not to create your own grid, or randomly generate one.

4.



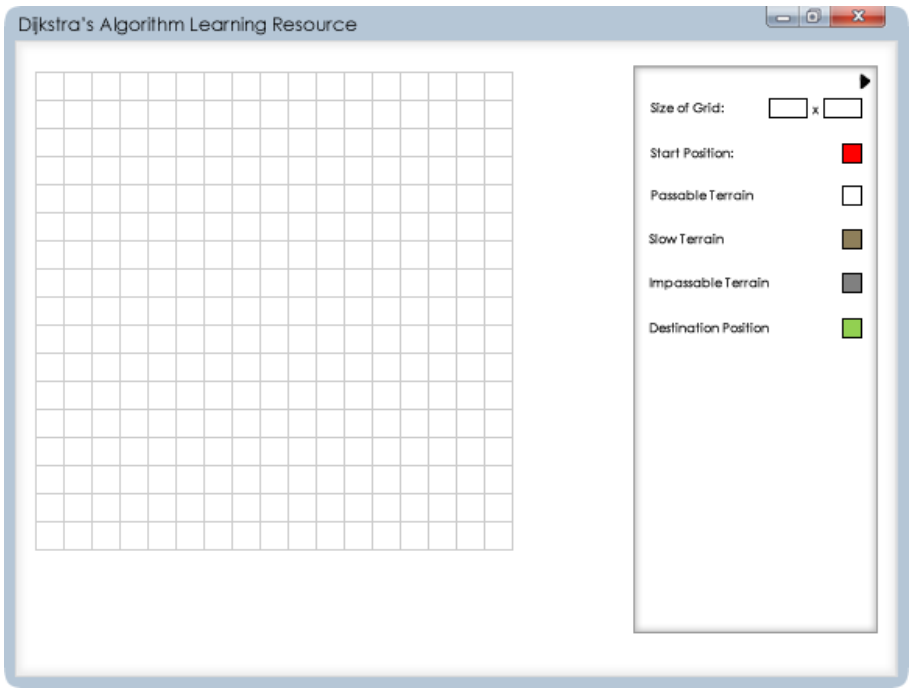
This option will appear if you decide to randomly generate a grid and you can choose parameters of the grid that will be created, you can decide the size of the grid and whether or not the grid generated will include cells with the slow terrain attribute. If you create your own grid the option for slow terrain won't be there and you will just choose the size of the grid.

5.



After you have chosen your options you will be presented with this screen, a blank grid with the options available in the top left and the solve button will be greyed out until a valid grid is available for Dijkstra's algorithm to be performed on.

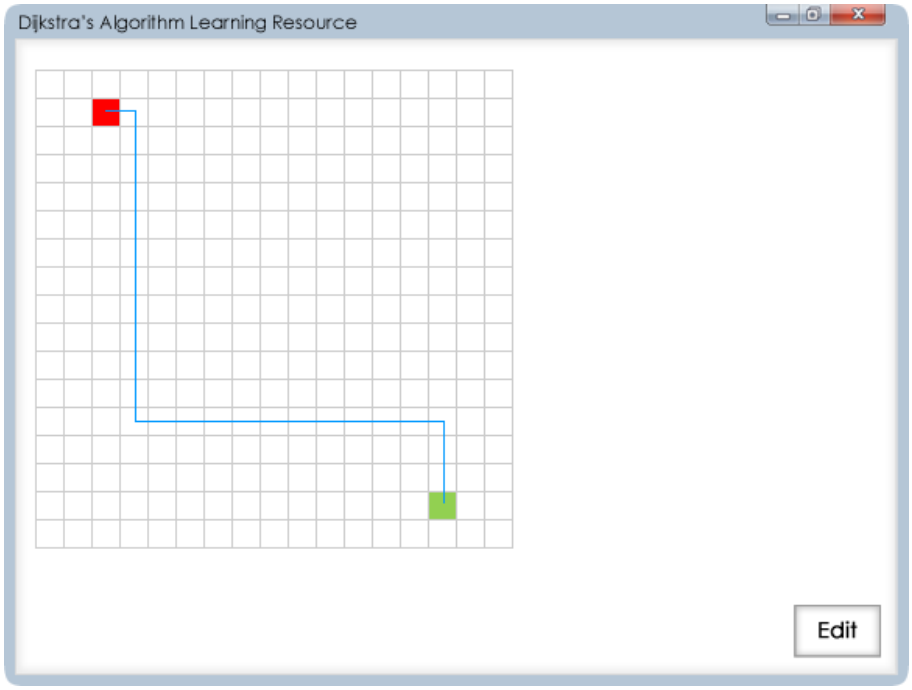
6.



After clicking the arrow, you are then presented with options to add attributes to the grid. If you have randomly generated a grid then you will be able to edit any attributes that have been generated. You can also change the size of the grid, if

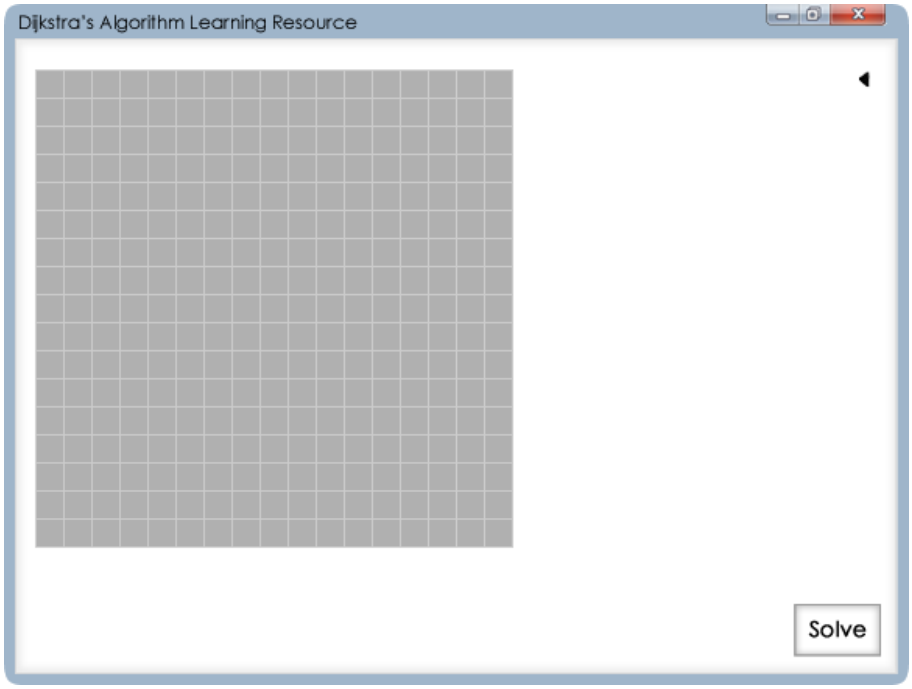
you make it smaller any cells with attributes will be lost and a warning will be displayed if the user is okay with this. To change an attribute you will click the coloured cell then be able to click any cells in the grid to apply that attribute to, if you are choosing the start or ending position and try to put more than one down, the old one will be removed when placing a new one.

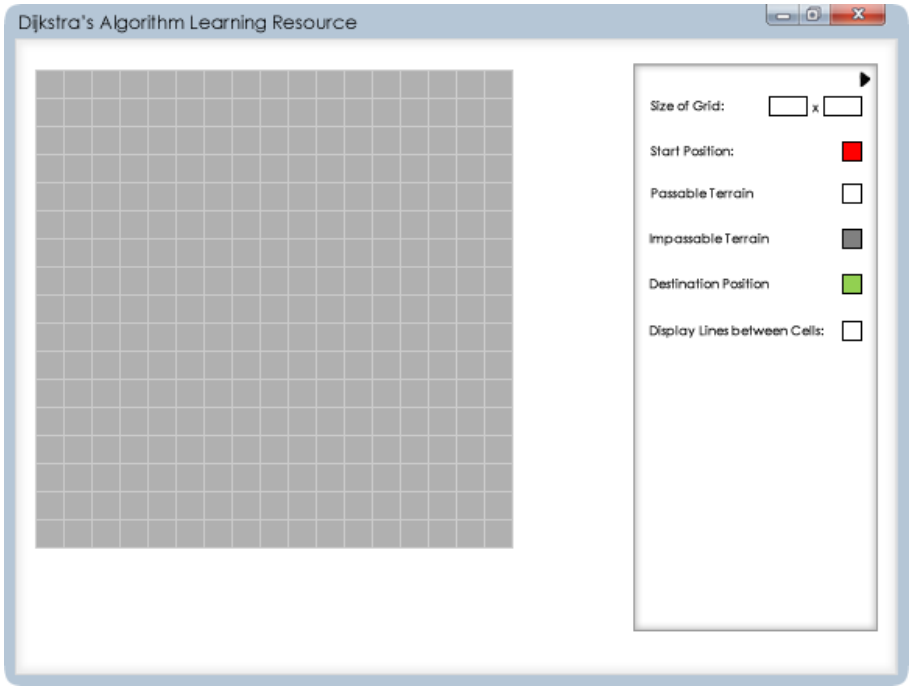
7.



If you have selected either options one a two, when you click solve a screen like this will appear where it will display, with a line, the shortest route from the start position to the destination position.

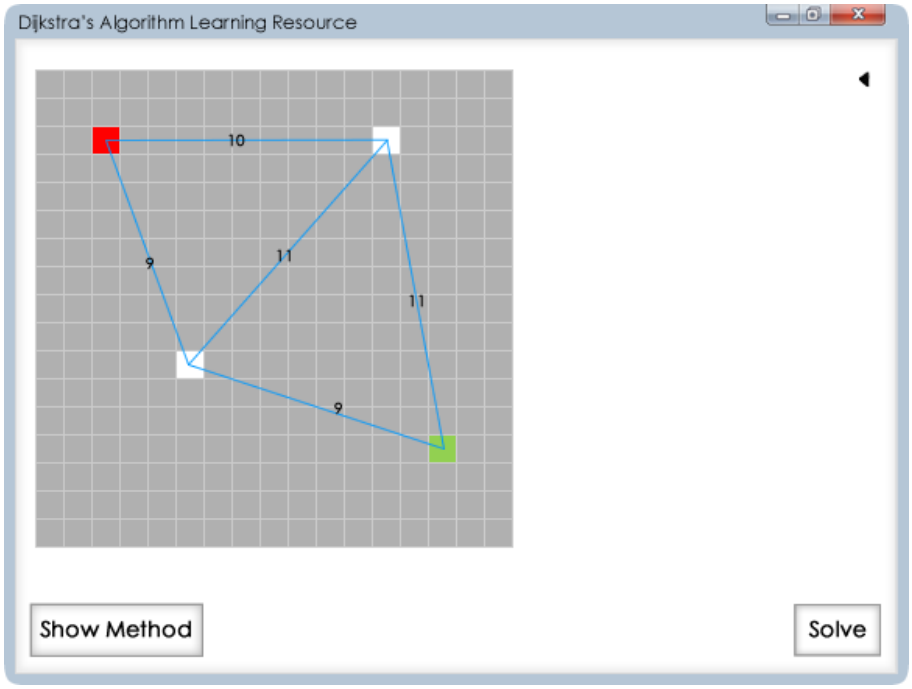
8.





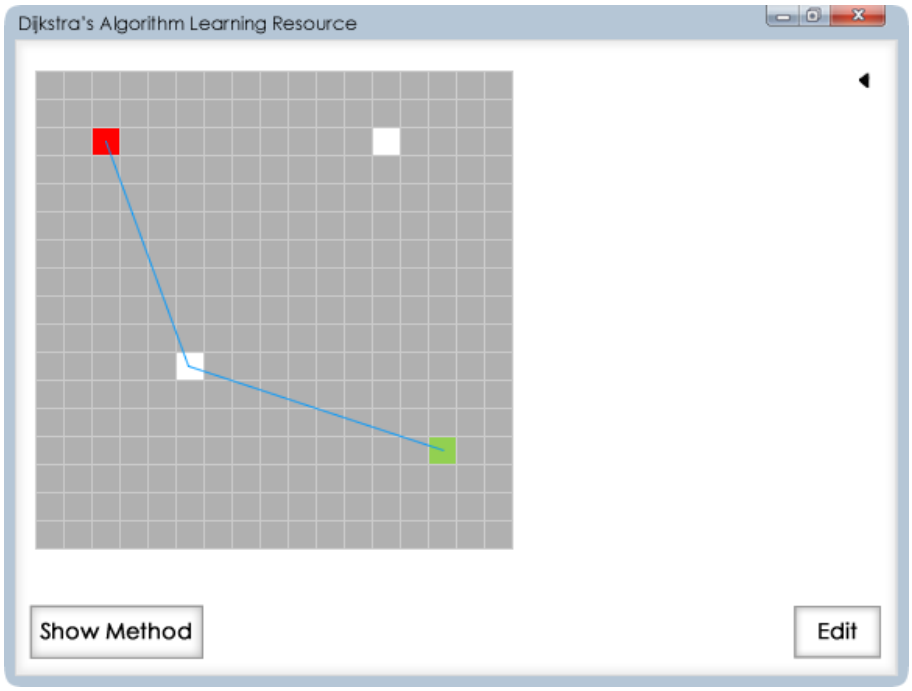
Here is what will be presented to you if you decide to choose option 3 in design 2, you will presented with a grid that is all impassable then choose cells that will be passable terrain. A passable terrain cell can't be within a certain range of another passable cell to give enough space to display the working. The program will then perform Pythagoras theorem to work out the distance between each cell and round it up or down to the nearest integer. The option "Display Lines between Cells" you can click to bring to a screen like the next design.

9.



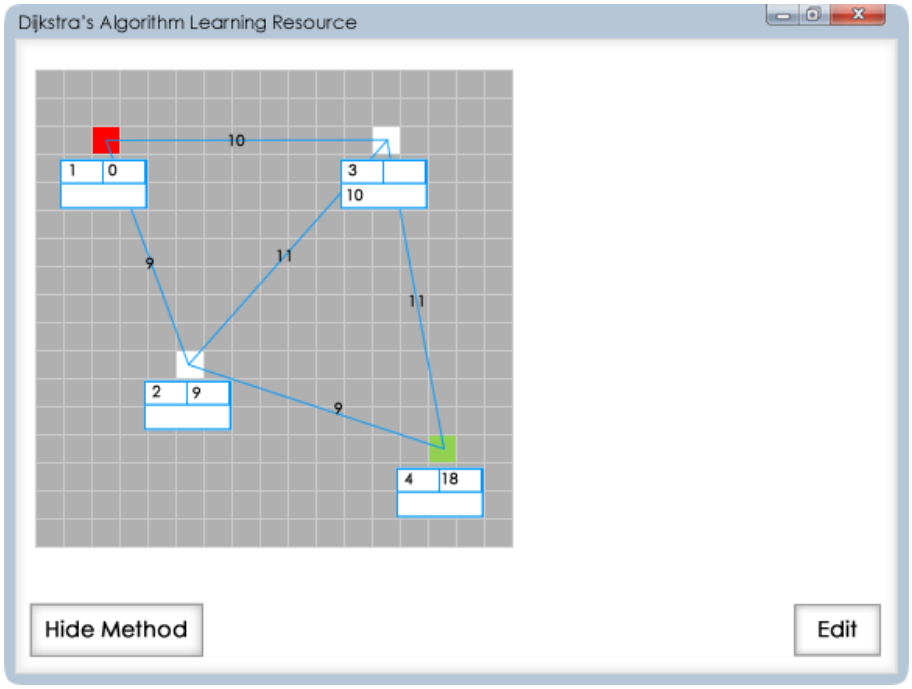
This is what it will look like if you select the "Shoe Lines between Cells" option. It will display the lines of working and distance between each cell and you will be able to double click on the numbers to be able to edit the distances.

10.



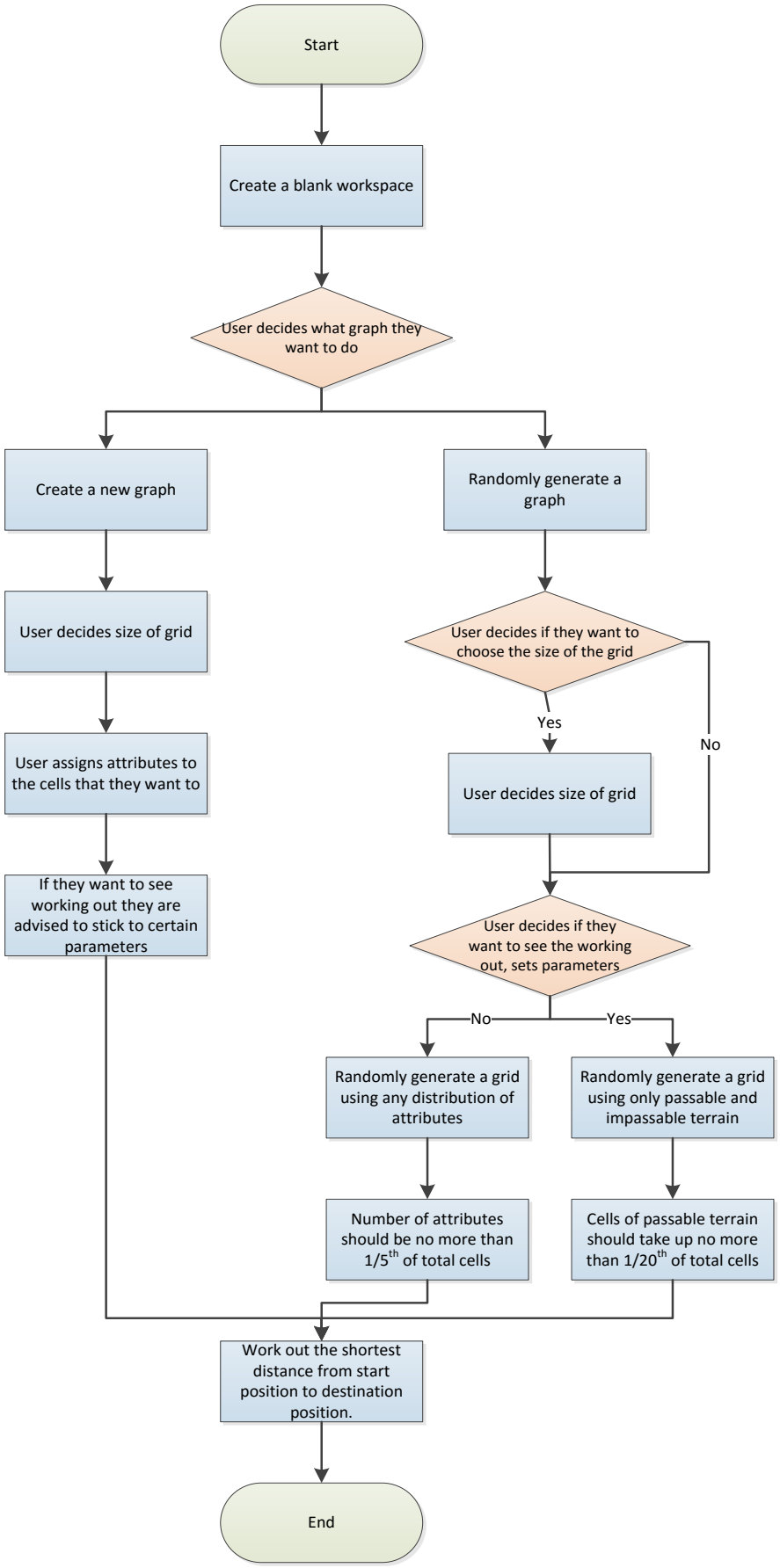
Once you click solve you will be presented with something like this, where it will show the shortest route from start to destination cell, you can then choose to show the method.

11.



If you choose to show the method, the lines and distances between cells will be shown again and the working out will be displayed; showing the permanent position, permanent values and temporary values of each cell.

System Flowchart



Security and Integrity of Data

By using the python extension it means that it will default to being opened in python so there's less possibility of opening it in the wrong program. When the program is opened it will check that all the files are suitable and that there are no errors, if any errors are found in the files, my program will display a suitable error method and show which file has been corrupted, this means that the program will not try and open with a corrupted file and potential cause further damage to the system. Furthermore by using an extension which is associated to Python it means that other programs are very unlikely to accidently use it and possibly overwrite important data. Lastly, by saving it in a shared network I can add security options, so if anyone wants to overwrite, edit or copy any of the data, they will have to make their own copy in their own drive which you can then select for use by the program so there is always a backup of the files.

Security of the System

To keep the program safe from any interference I will employ a couple of different security features. Firstly I will compile the program into an executable file (.exe) so that any normal users won't accidentally open the program in editing software by mistake. Also I will make it so if they do open the file in Pyscripter, it will be in compatibility mode, where the user can look at and copy the code, however to edit it they first have to make a copy in their own drives. Furthermore any important files, such as text files which contain important data stored elsewhere or any other important aspects of the program will be stored in a separate folder which will require administrator privileges and a password to access.

Test Strategy

I will use a couple of different strategies and techniques to test the structure and functionality of my program. Firstly I will perform a self-assessment of my code. I will employ three different techniques to make sure I fully test my system, these are: bottom-up testing; white box testing and black box testing. I will then get peers to do some black box testing before finally getting students and teachers of my target audience to test the usability of the program.

Bottom-Up Testing

This method of testing works by looking at the smaller modules and functions in a program first and then works up, basically working from the bottom to the top of the hierarchy of functions. This method is good since it will allow me to find any errors in any of the functions first and then test how these functions will interact with one another.

Black Box Testing

This method of testing tests the functionality of the program, rather than its internal structures or workings. It will test that the program or application gives out the right

output when given a specific input, without worrying about how it gets from the input to the output. Due to the tester not needing to know how to get from the input to the output, this is a suitable testing method for my peers to use.

White Box Testing

White box testing is the opposite of black box testing, rather than looking at the functionality of a program it just looks at the internal structures and workings. In white box testing the tester takes part of the code and using their programming skills and internal perspective of the system designs test cases for that part of the code. The tester then chooses an appropriate set of inputs and looks at how the program gets to the outputs it does, to make sure there is consistency in the code and that it will work again and again. Due the internal perspective and programming skills required, unless I can find someone willing to spend hours looking at the code of my program to understand it, I will be doing this method of testing myself.

Testing

Test Plan

When conducting my tests I will be using these test types:

- Normal - Data that is expected to be entered into that field
- Extreme - Data that is at the very edge of the accepted data, putting 1 or 10 in a field that accepts numbers between 1 and 10
- Erroneous - Data that is expected to give an error in that field

Most of my testing is done using the Black Box testing method. I will test the functionality and robustness of the program and when I encounter any errors I employ other testing methods appropriately depending on the error.

Test Set	Test	Expected Results
1	Menu buttons	When the buttons are clicked, it sends it to the next appropriate screen. After the options screen the grid should be displayed with customised options depending on the function.
2	Options screen inputs	You can enter a number and click outside the box, or press enter, to input the data. If the number is larger or smaller than accepted values bring it to the closest appropriate value. If left blank, display previous number.
3	Changing the size of the grid	The grid should update as soon as new validated value has been entered and an appropriate size for each cell decided.
4	Editing the cells of the grid	When a cell type is selected and a cell is clicked on within the grid, that cell's properties should be updated with the selected cell type as long as it is valid.
5	Adding and clearing lines on the grid	Adding a line should display it on screen with the length to 2 significant figures, calculated using the Pythagoras theorem, displayed in the middle of the line. Lines shouldn't be drawn when they intersect with another line and cell editing options disabled whilst lines are on the grid.
6	Solving the graph	The shortest route is displayed over the graph if there is one; if the graph is invalid do not solve the graph.

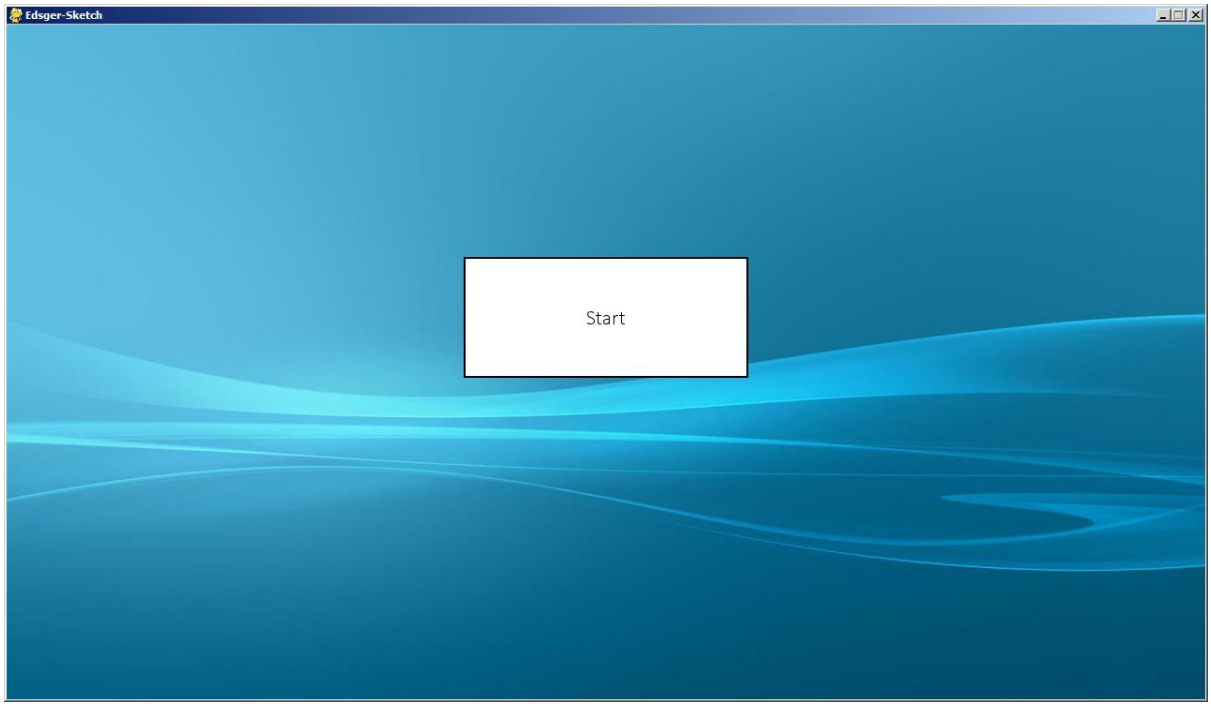
Test Results

Test Set 1

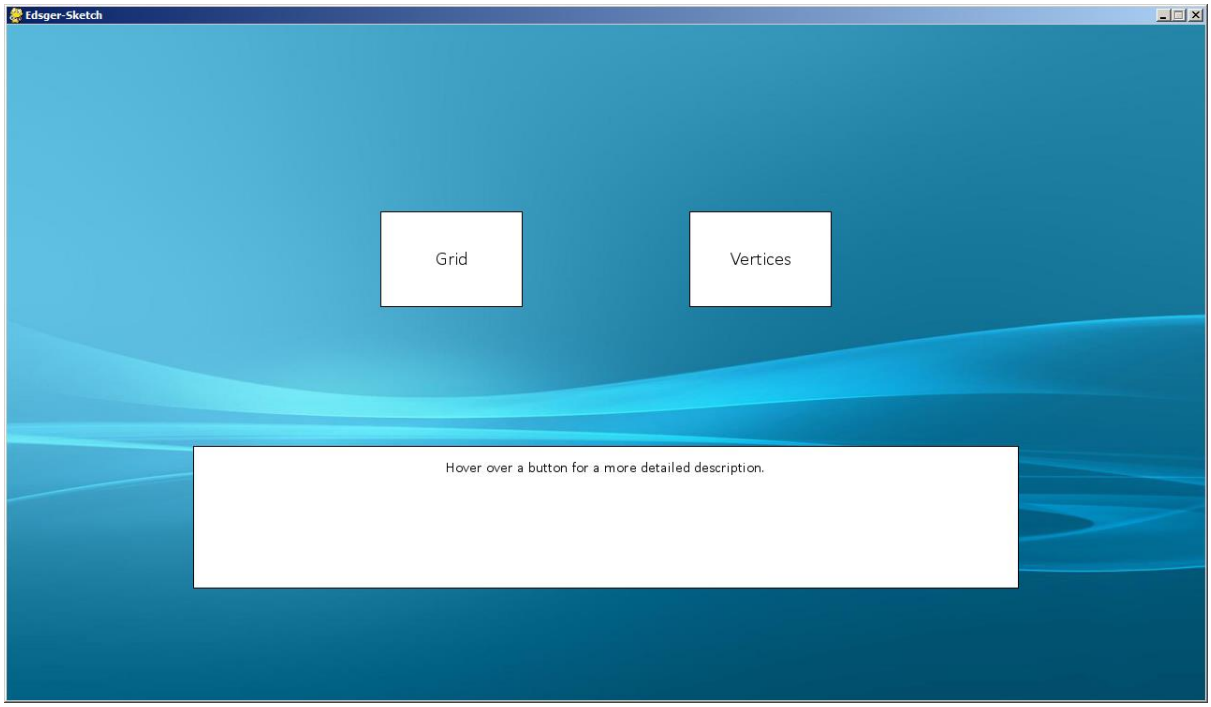
Here I am testing all the buttons that link from the main screen to the functions and the pages in between.

Tested Data	Test Type	Expected Result	Achieved Result
Testing that the menu buttons function accordingly when clicking on them	Normal - Clicking within the button	Button should register click and takes me to the next page	Button registered click and took me to the next page
	Extreme - Clicking the outline of the button	Button shouldn't register click and keep me on the same page	Button didn't register click and kept me on the same page
	Erroneous - Clicking outside the button	Button shouldn't register click and keep me on the same page	Button didn't register click and kept me on the same page
Testing that the menu buttons function accordingly when hovering over them	Normal - Hovering within the button	Button should register the mouse is on the button and display appropriate data	Button registered the mouse was on the button and displayed the appropriate data
	Extreme - Hovering on the outline of the button	Button shouldn't register the mouse is on the button and keep the original data displayed	Button didn't register the mouse was on the button and displayed the original data
	Erroneous - Hovering outside the button	Button shouldn't register the mouse is on the button and keep the original data displayed	Button didn't register the mouse was on the button and displayed the original data

When starting the program this is the first page to appear:

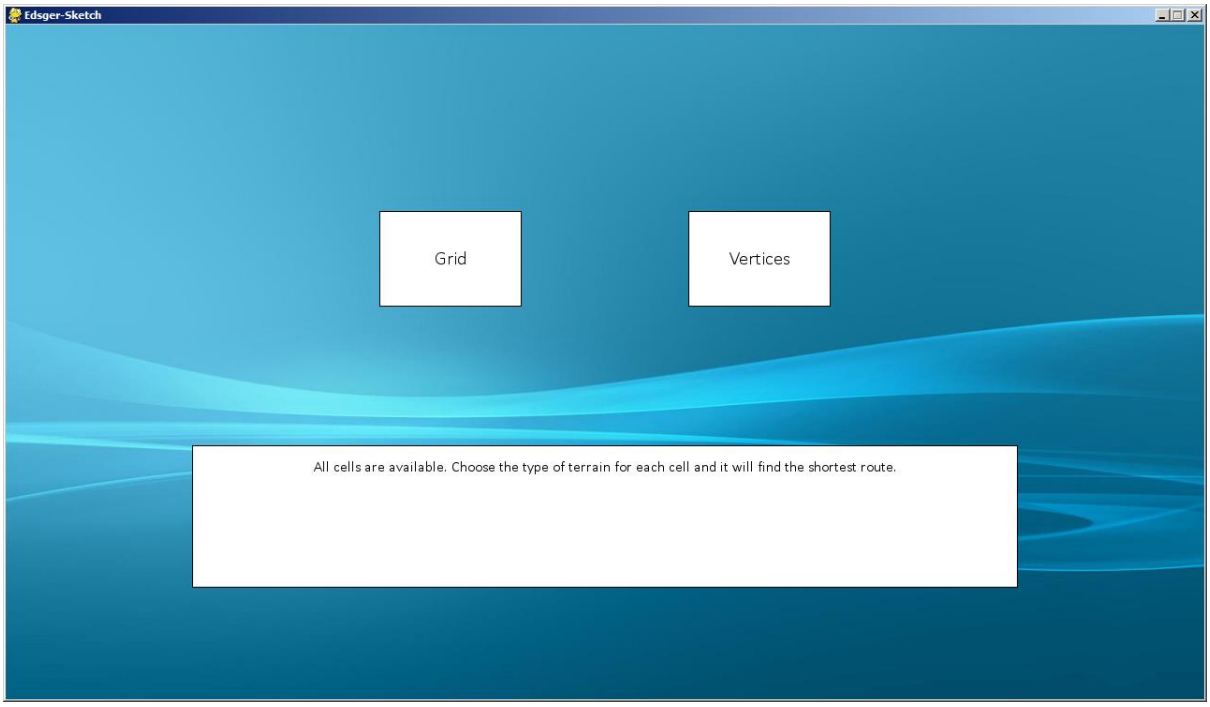


Clicking the start button on the main screen takes me to the page where I can choose which function I want to use:

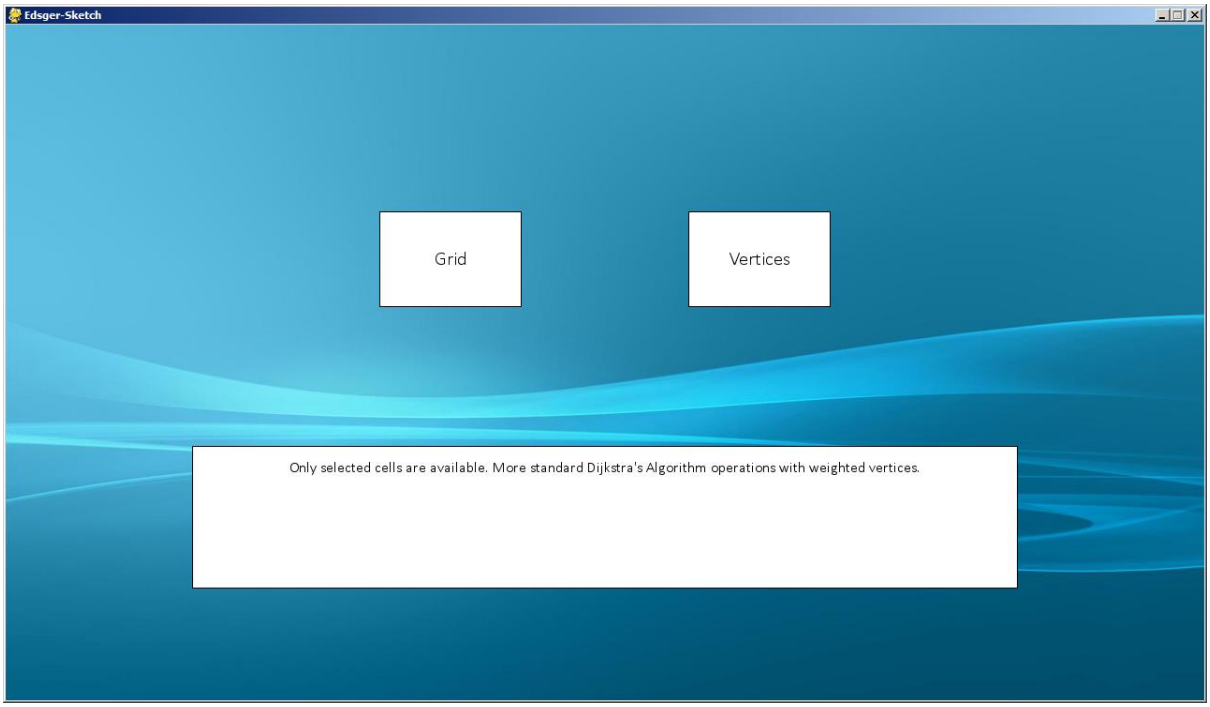


Clicking either outside the button, or on the black outline of the button, results in nothing happening and the same page, with the start button, still being displayed.

On the choose function screen, hovering over the Grid button results in the following text being displayed inside the text box:

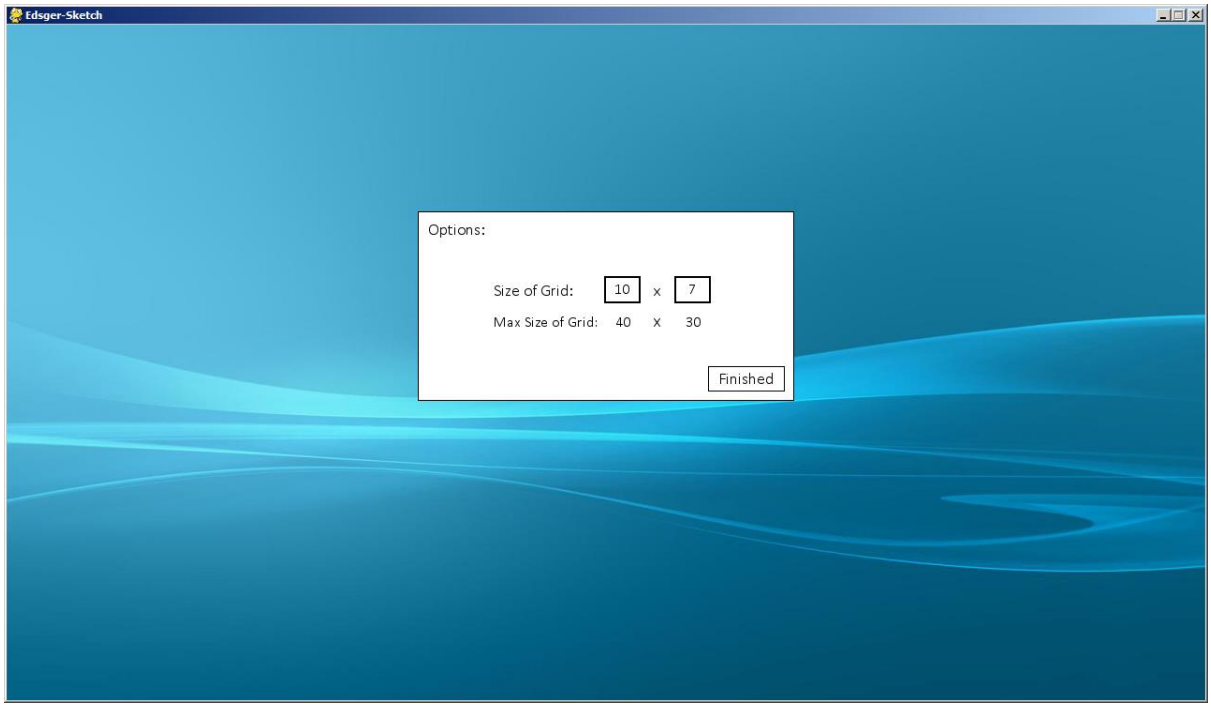


Hovering over the Vertices button results in this text being displayed in the text box:

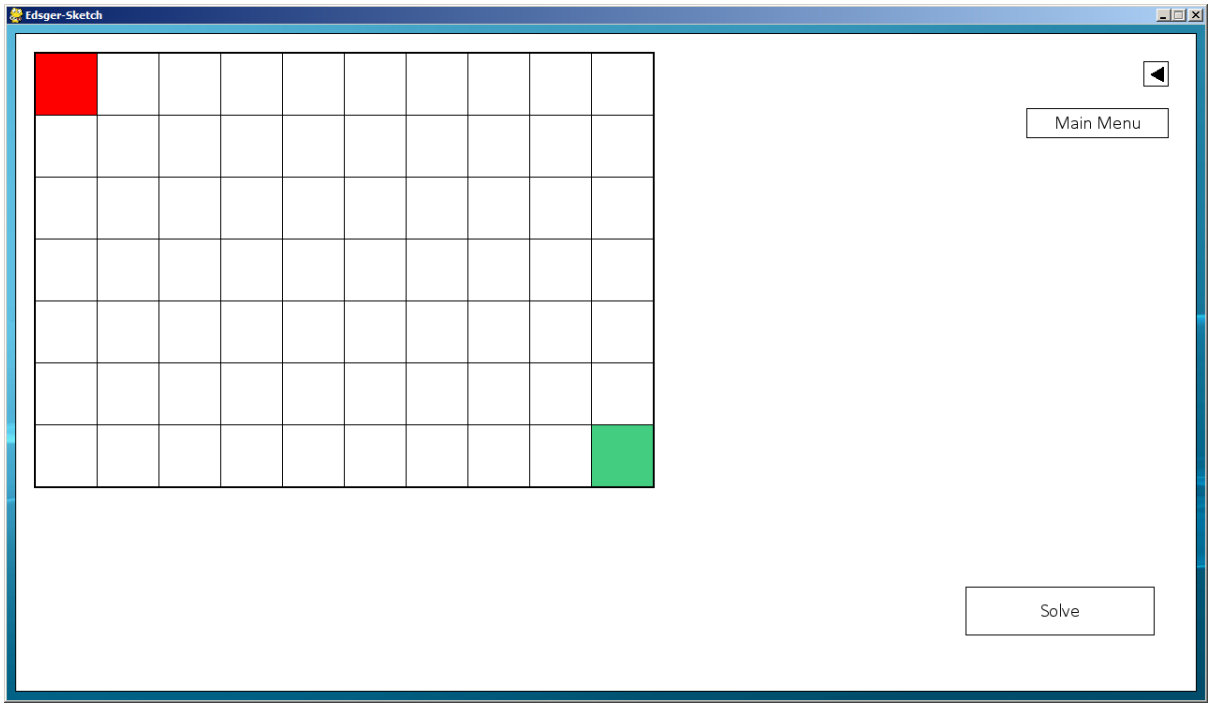


If I am not hovering over a button, hovering over the text box itself, or hovering over the black outline to the box, the text box displays the original text "Hover over a button for a more detailed description."

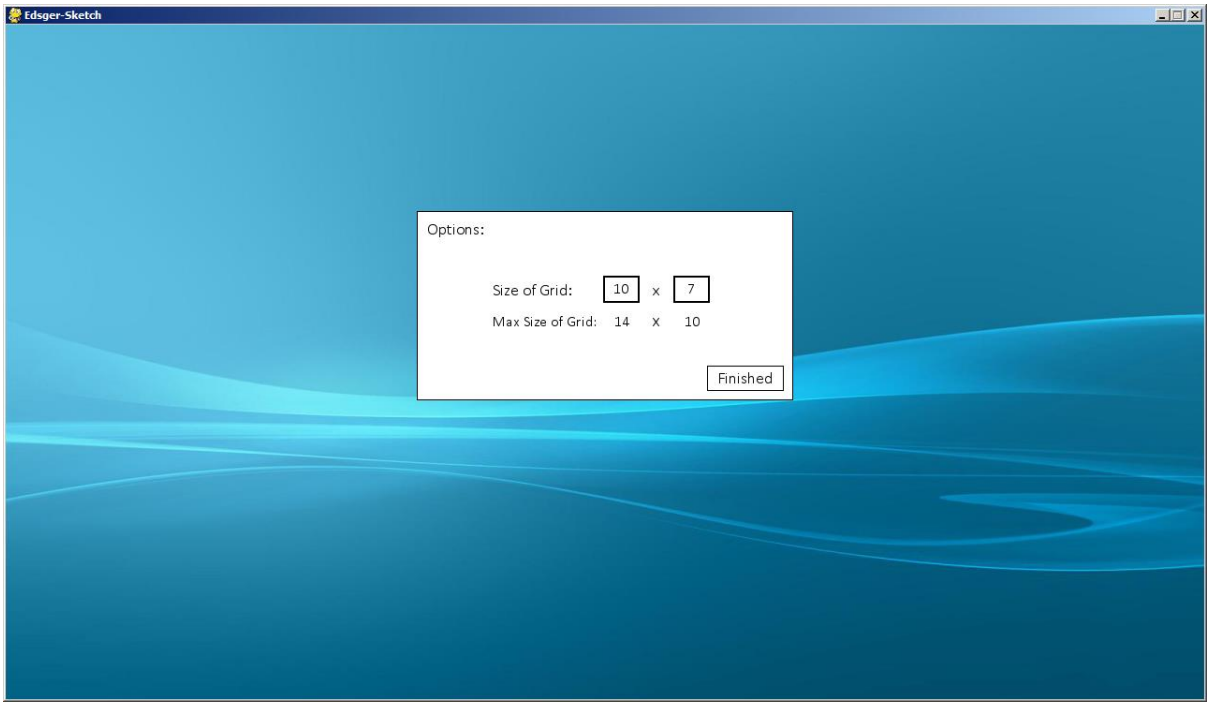
When clicking either function, you are sent to the options page where you can change the size of the grid, however the max size of grid is updated accordingly, this is the options page when clicking the Grid button:



Clicking the finished button takes you to the final page with appropriate grid being displayed:

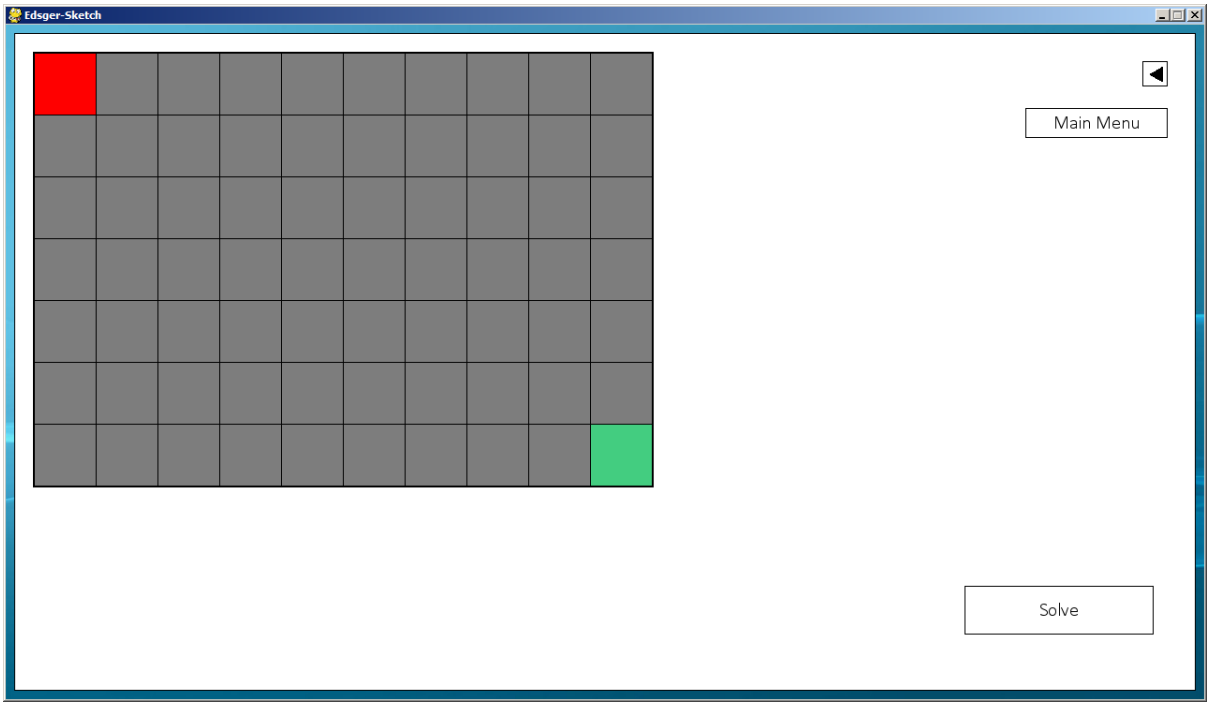


Clicking the Vertices button on the choose function page results in this page being displayed:



The max grid size is updated appropriately for the vertices function.

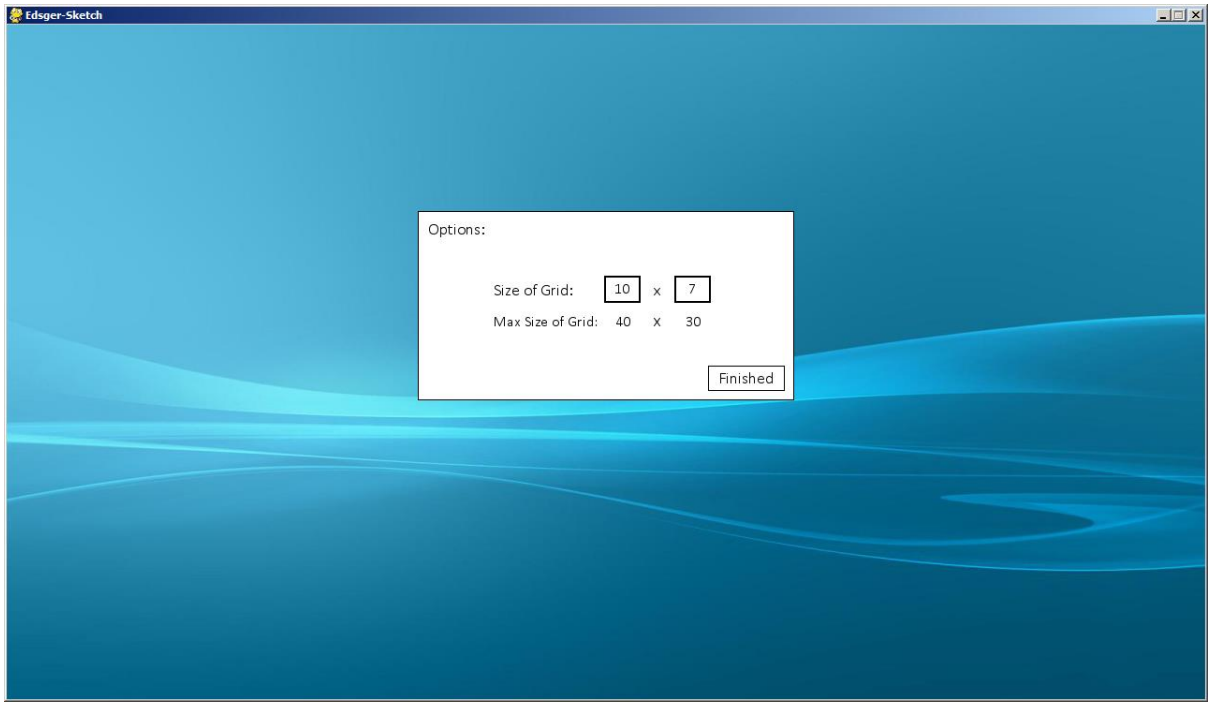
Once finish has been clicked the next screen is displayed with a grid size matching the users input, for this all the cells are set to be unpassable terrain:



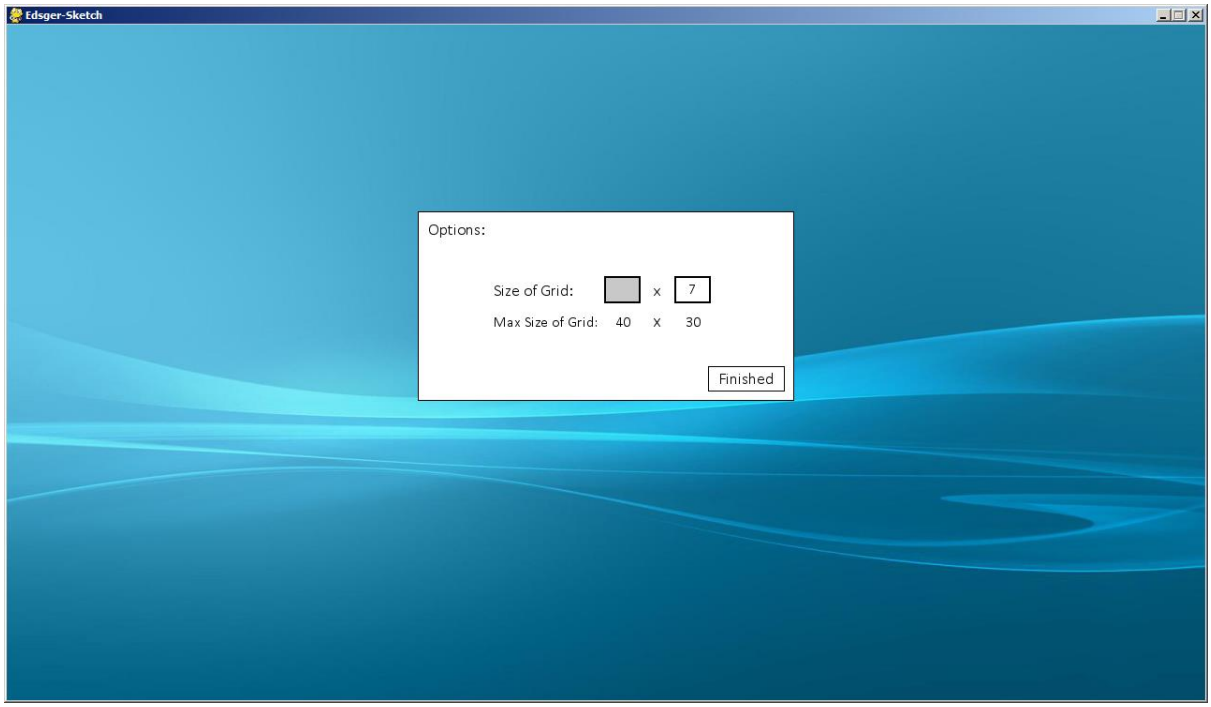
Test Set 2

Tested Data	Test Type	Expected Result	Achieved Result
Testing the text inputs function correctly when clicking on them	Normal - Clicking within the input box	The text input should be shaded grey to show it is selected and further inputs should be displayed	Expected result is achieved
	Extreme - Clicking the outline of the input box	Input box shouldn't register the click and not take inputted data	Expected result is achieved
	Erroneous - Clicking outside the input box	Input box shouldn't register the click and not take inputted data	Expected result is achieved
Testing that you can easily exit the text box and the inputted data is validated	Normal - Clicking outside the box/	The input should be validated and the appropriate value displayed and the user shouldn't be able to enter any more data	Expected result is achieved
	Extreme - Clicking the outline of the box	The input should be validated and the appropriate value displayed and the user shouldn't be able to enter any more data	Expected result is achieved
	Erroneous - Clicking within the box	The input should stay within the box and no actions should be taken	Expected result is achieved

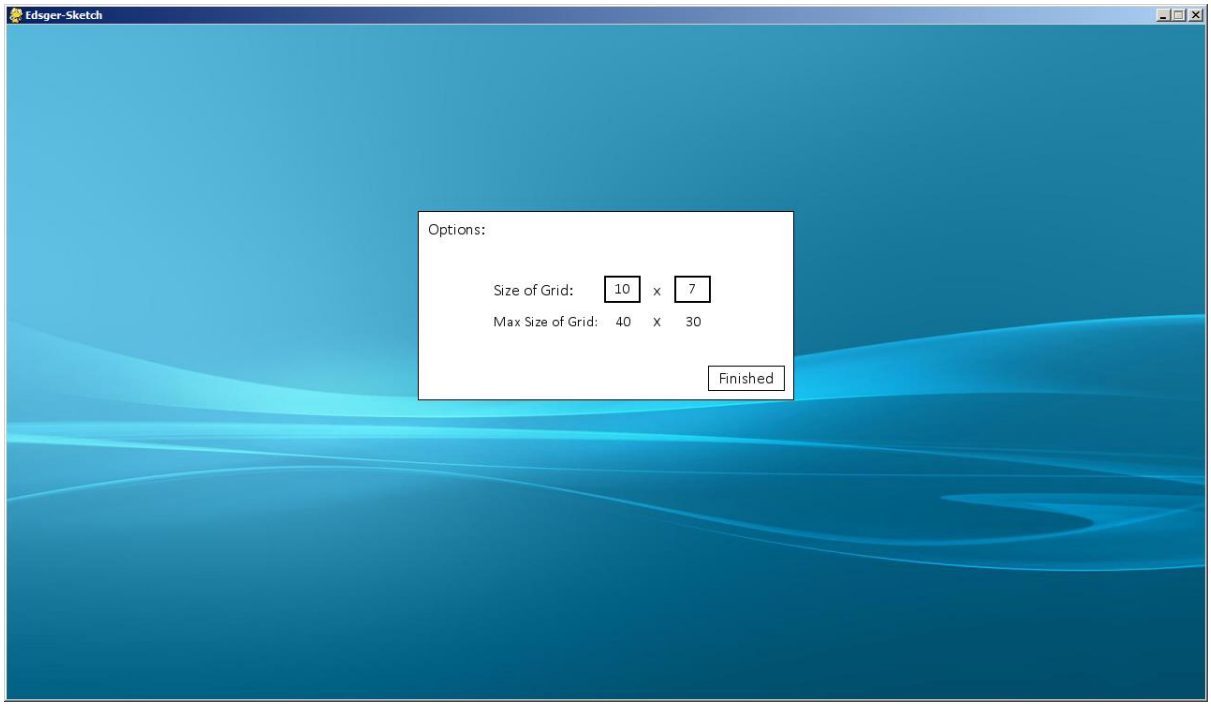
This is the screen I am going to be testing:



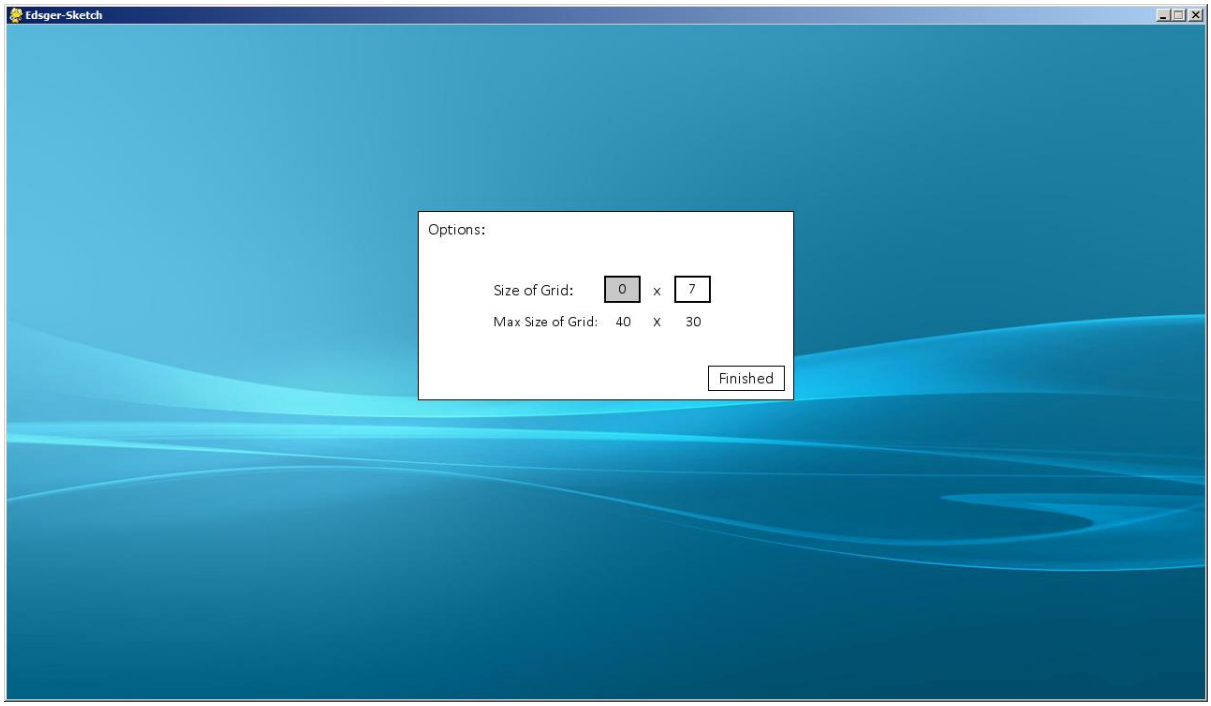
When trying to enter any characters which aren't the numbers 1 to 10, it doesn't accept the input and leaves it blank. Here I leave the input blank:



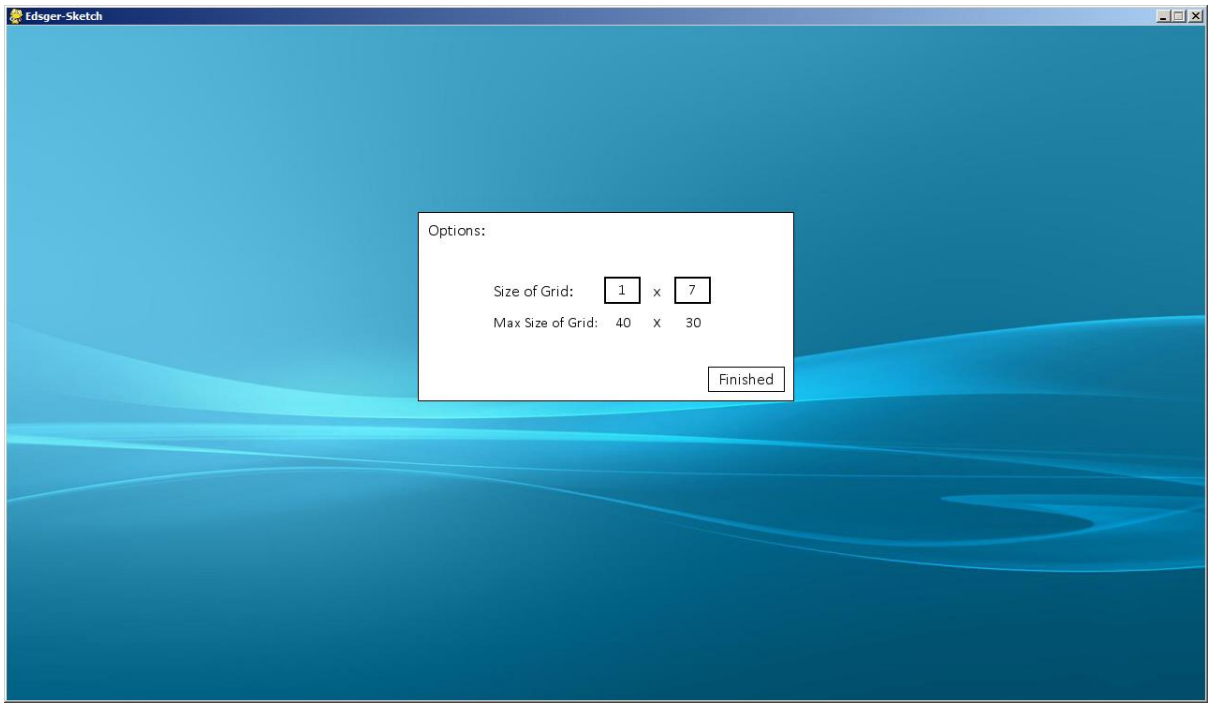
Since there is valid data it simply returns the original value of 10:



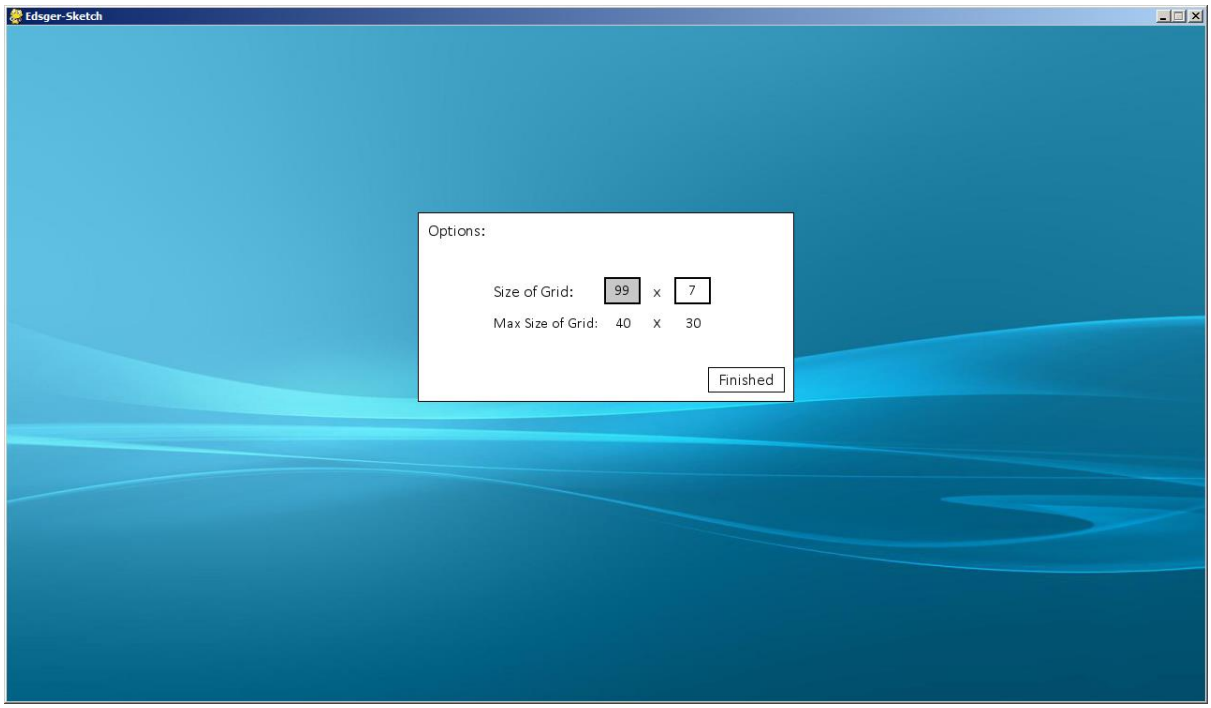
Here I choose 0 as a value



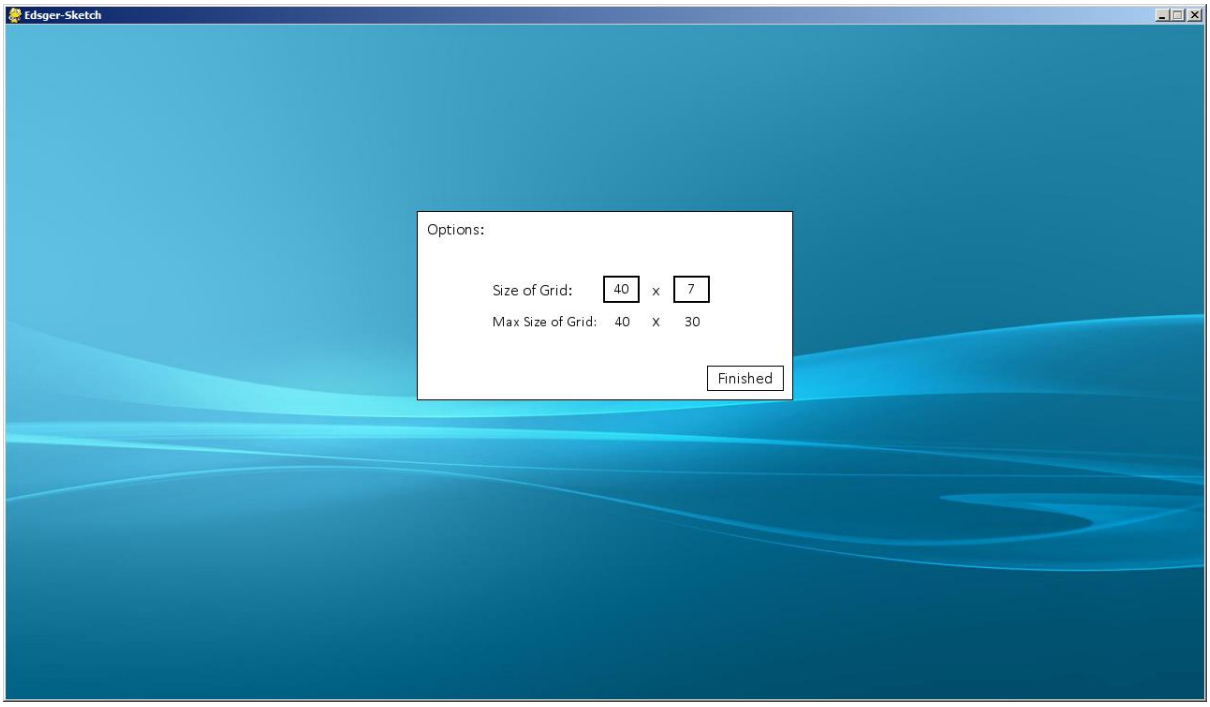
Since the graph cannot be 0 cells along, it simply replaces it with the smallest available value, which is 1:



Since the max number is 40, which is two digits, if I try and input any more than 2 digits, it doesn't accept any further inputs. Here I try entering the highest number possible, which with this validation, is 99:

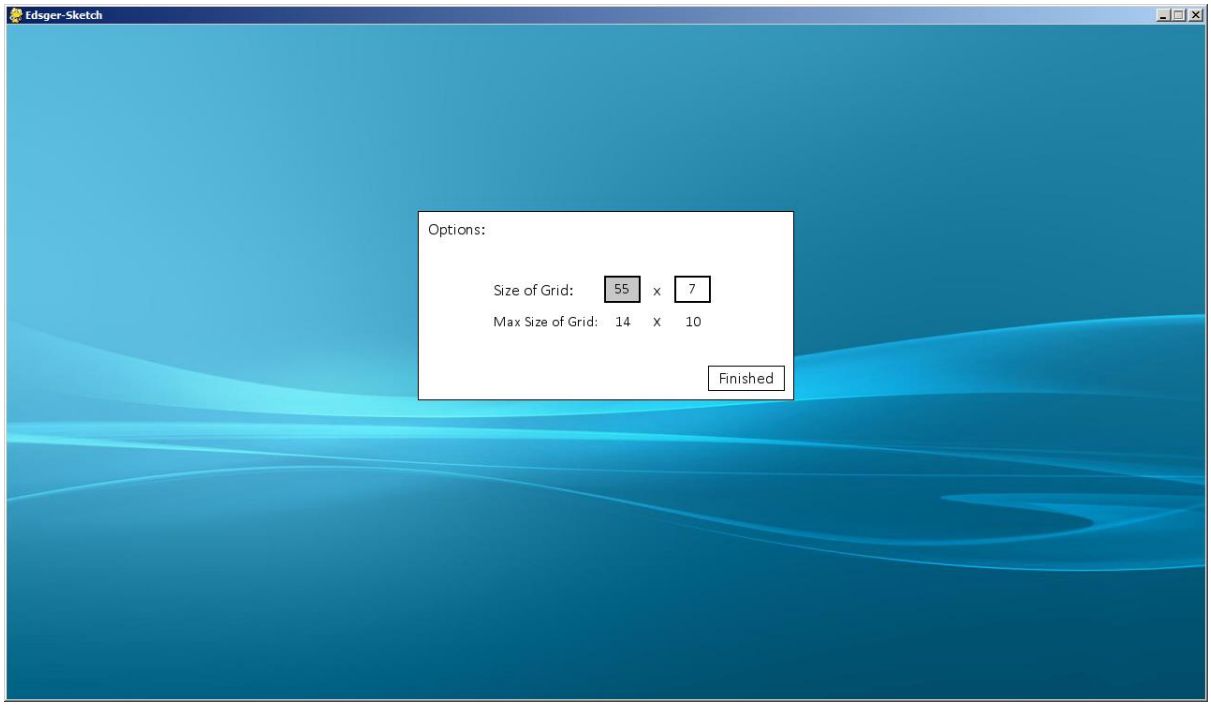


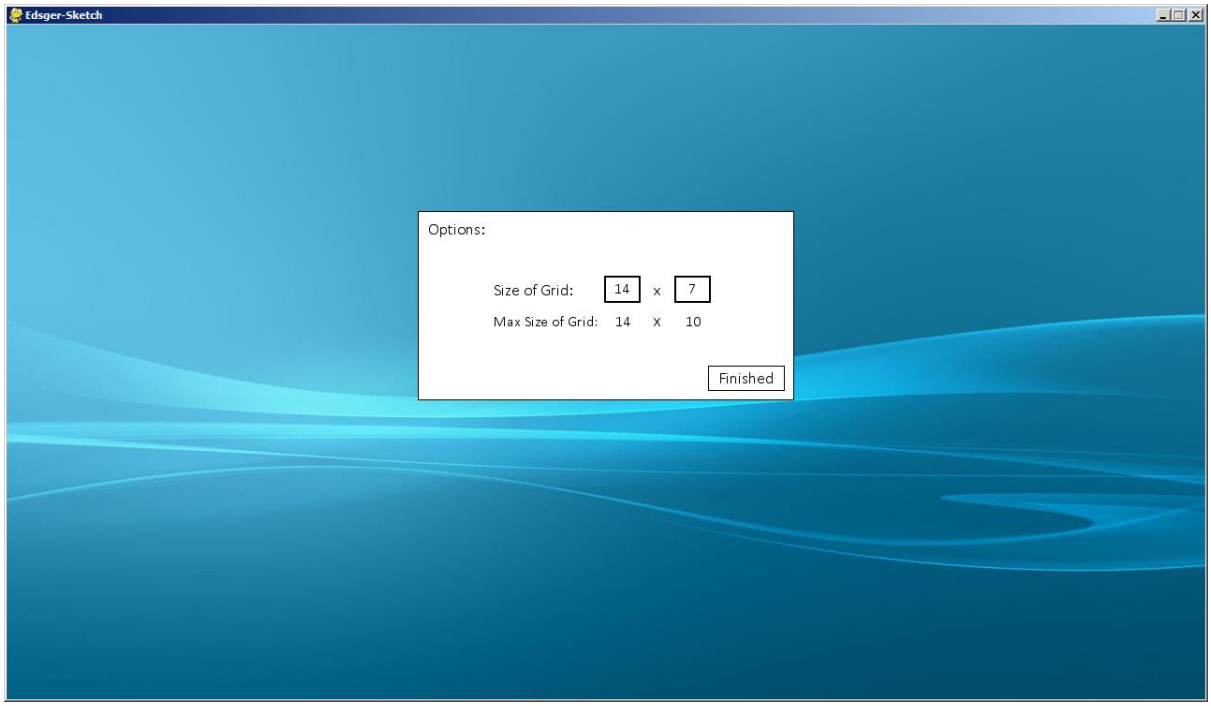
Since it is higher than the accepted value, it returns it as the highest accepted value, which for function 1, the Grid function, is 40:



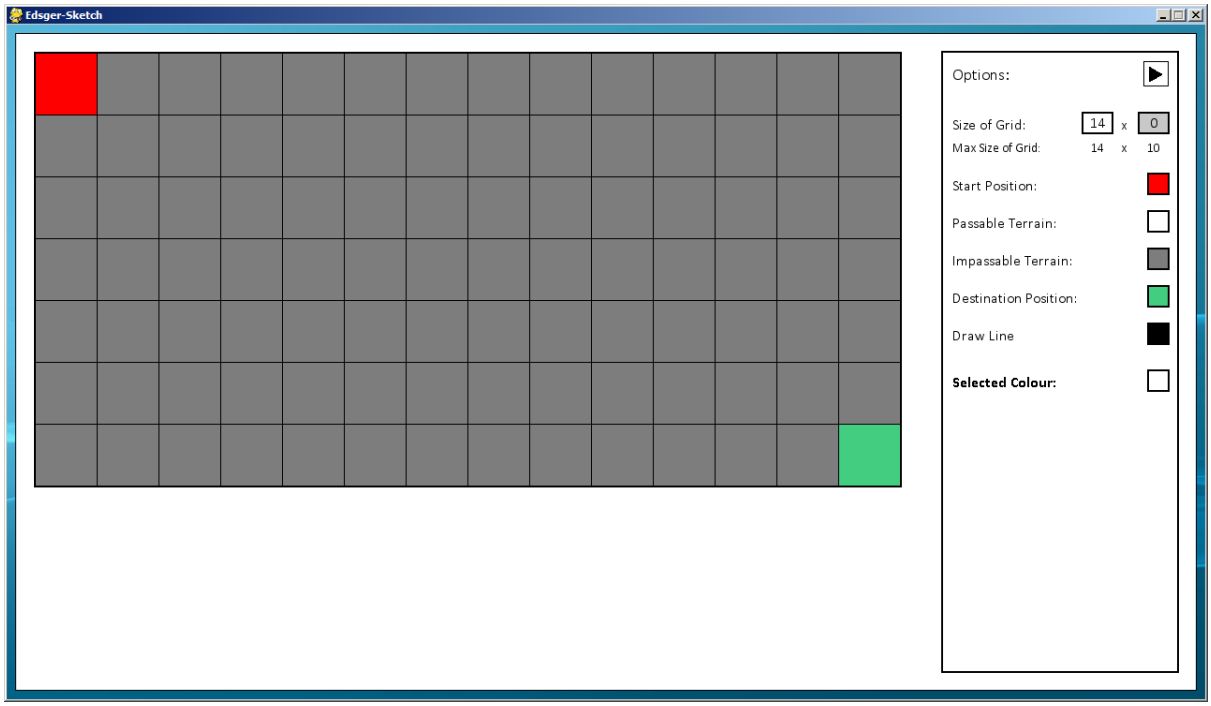
All these tests were conducted on the other input areas as well; here is a quick overview of the other two areas these inputs appear:

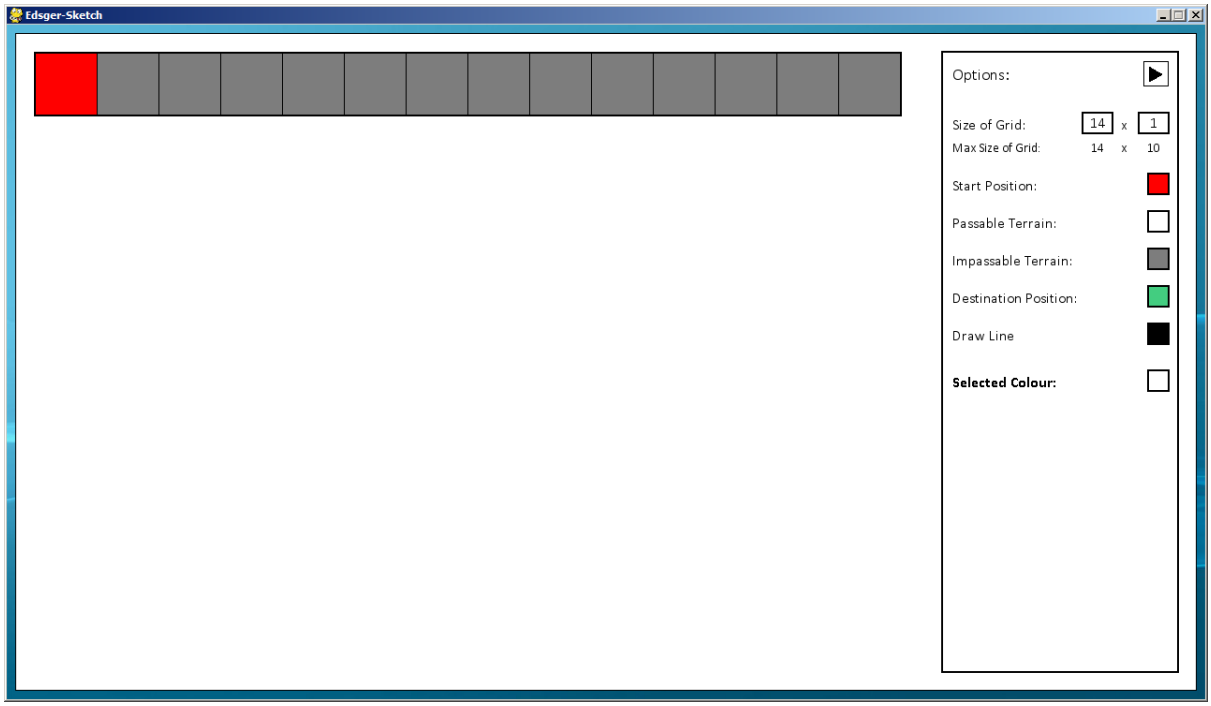
This is the vertices function:





The side bar on the main page of the program:

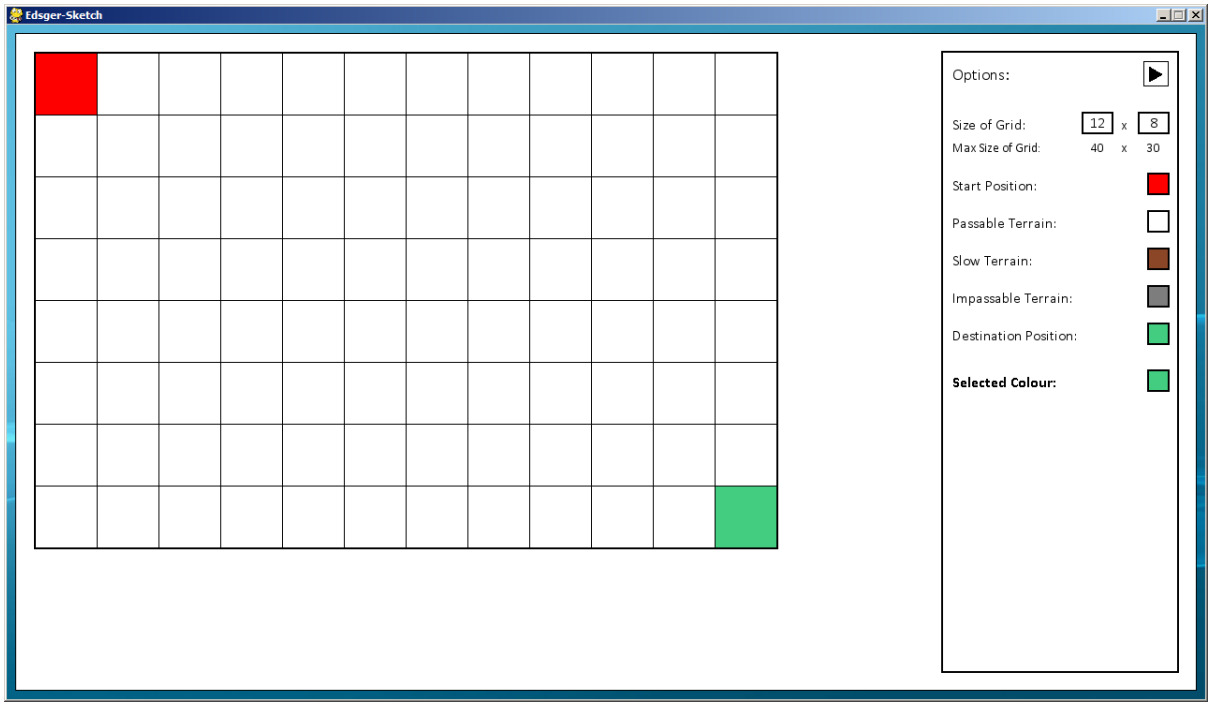




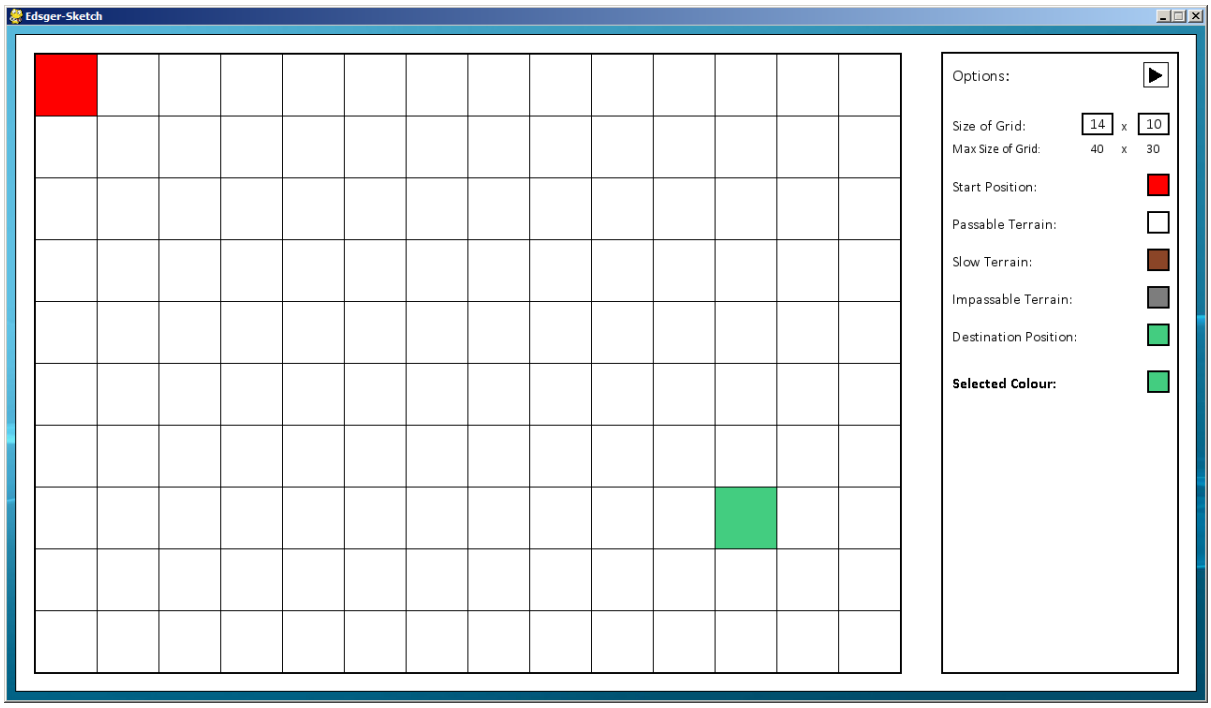
Test Set 3

Tested Data	Test Type	Expected Result	Achieved Result
Testing that the grid updates appropriately with the grid size	Normal - Entering an accepted value	The grid should update with the cells being an appropriate size	Expected result is achieved
	Extreme - Entering the minimum/ maximum accepted values	The grid should still update, which the cells being an appropriate size	Expected result is achieved
	Erroneous - Entering a value that isn't accepted	Validation should take place, as shown in the previous set of tests	Expected result is achieved

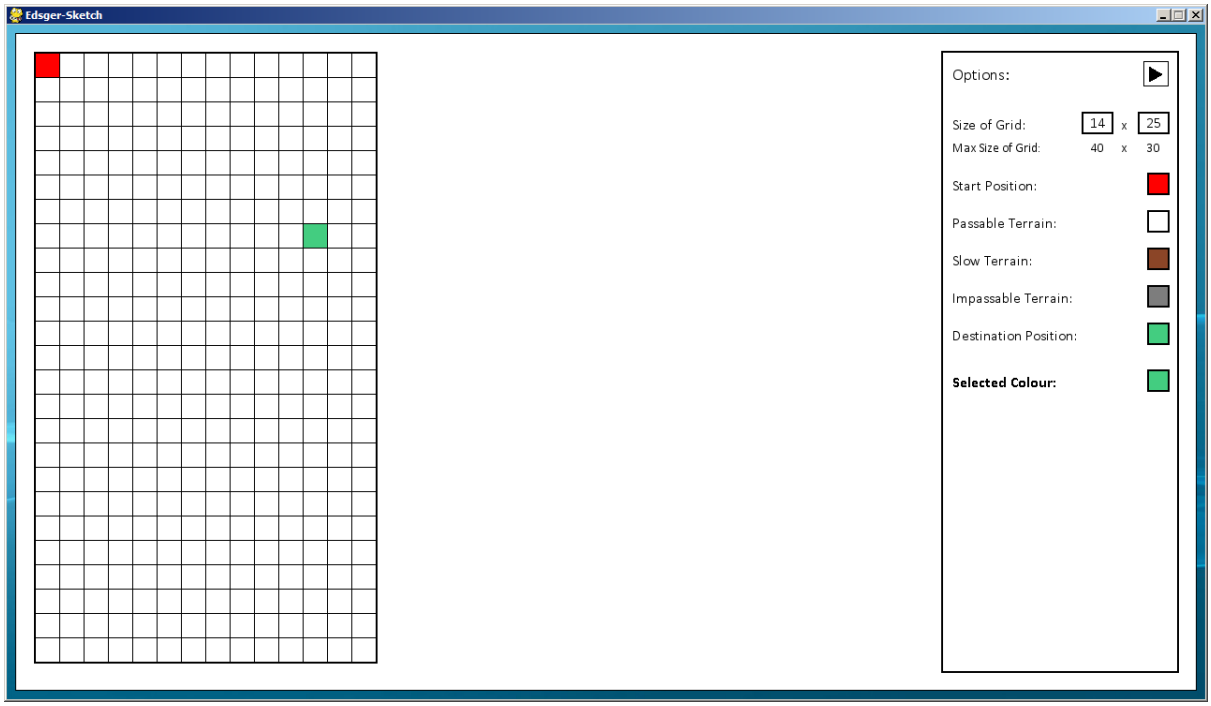
Here I am functions screen I am going to be testing, this is the Grid function. First off, I start with a grid size that is valid and uses the basic cell size of 66 pixels.



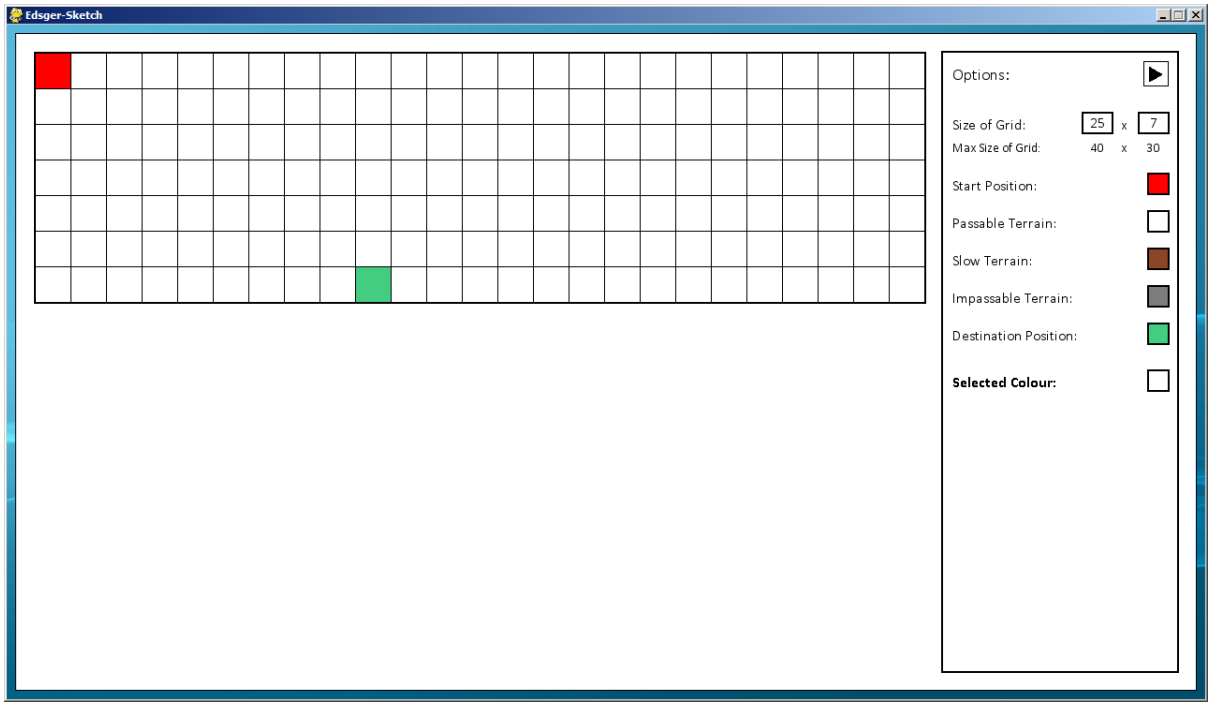
I then change this to the maximum grid size that can still use the default cell size of 66 pixels.



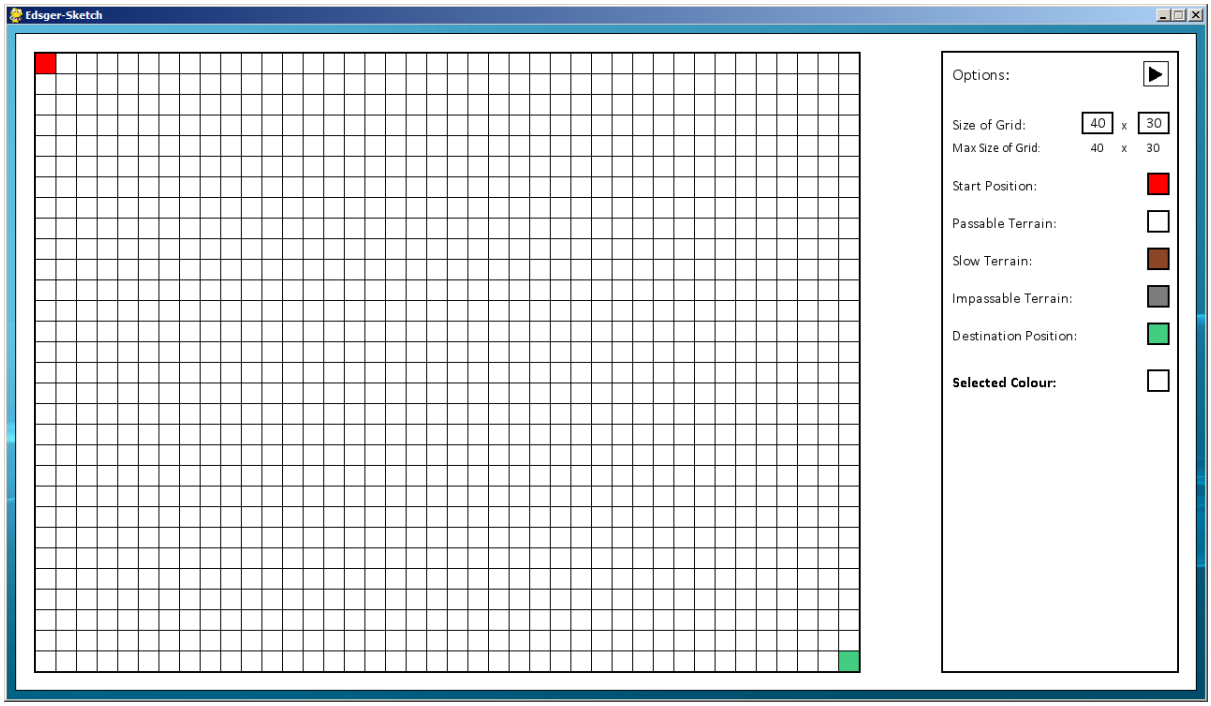
I then test that, when going above this grid size of 14x10, that the cell size should update accordingly, depending on which value is the limiting factor. The cell size is calculated by taking the number of available pixels in each axis and dividing by the number of cells, whichever is the smallest value is the limiting factor.



I then test the x value

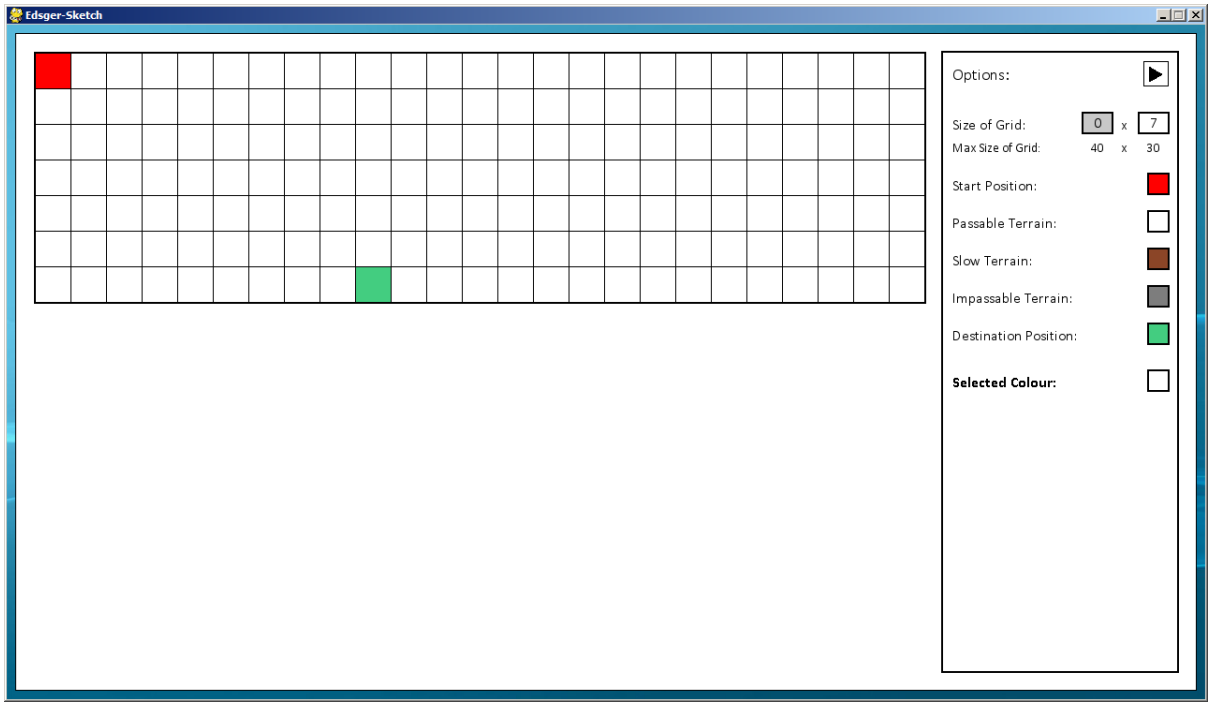


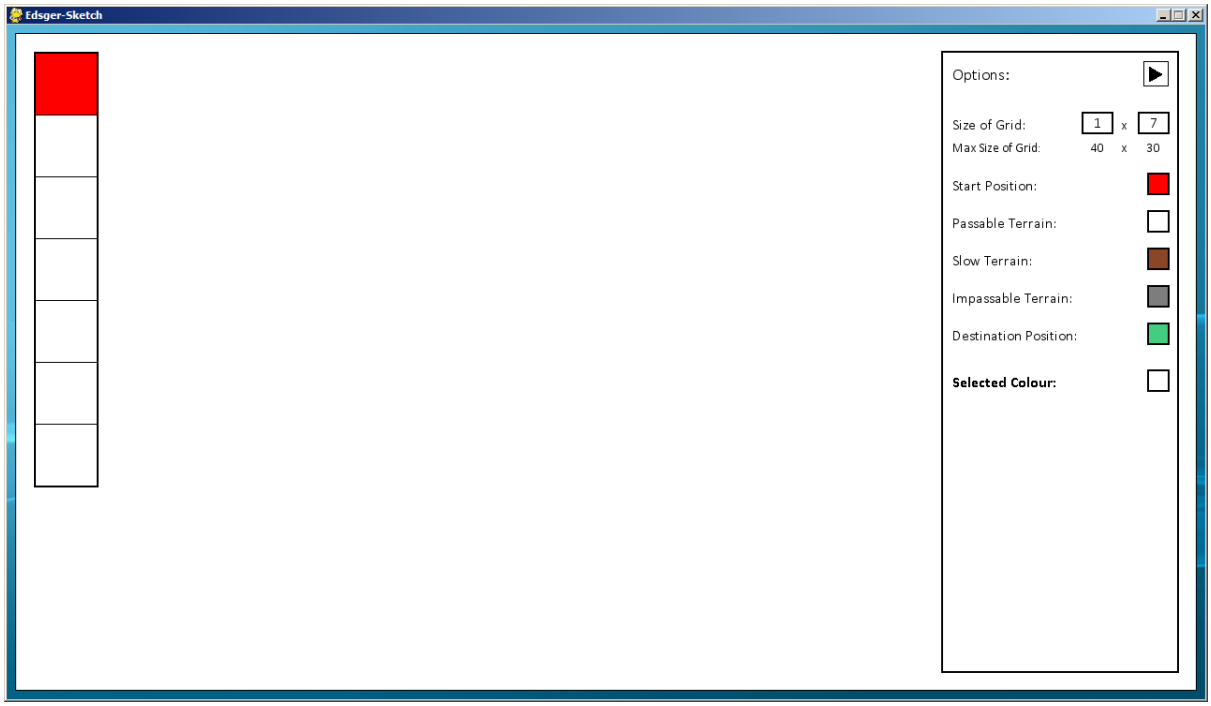
Both values work fine so I tested using both the maximum values



The grid updates to a cell size limited by the y axis, since that is the axis which is reaching the edge of the screen.

I then test what happens when entering a value too small:



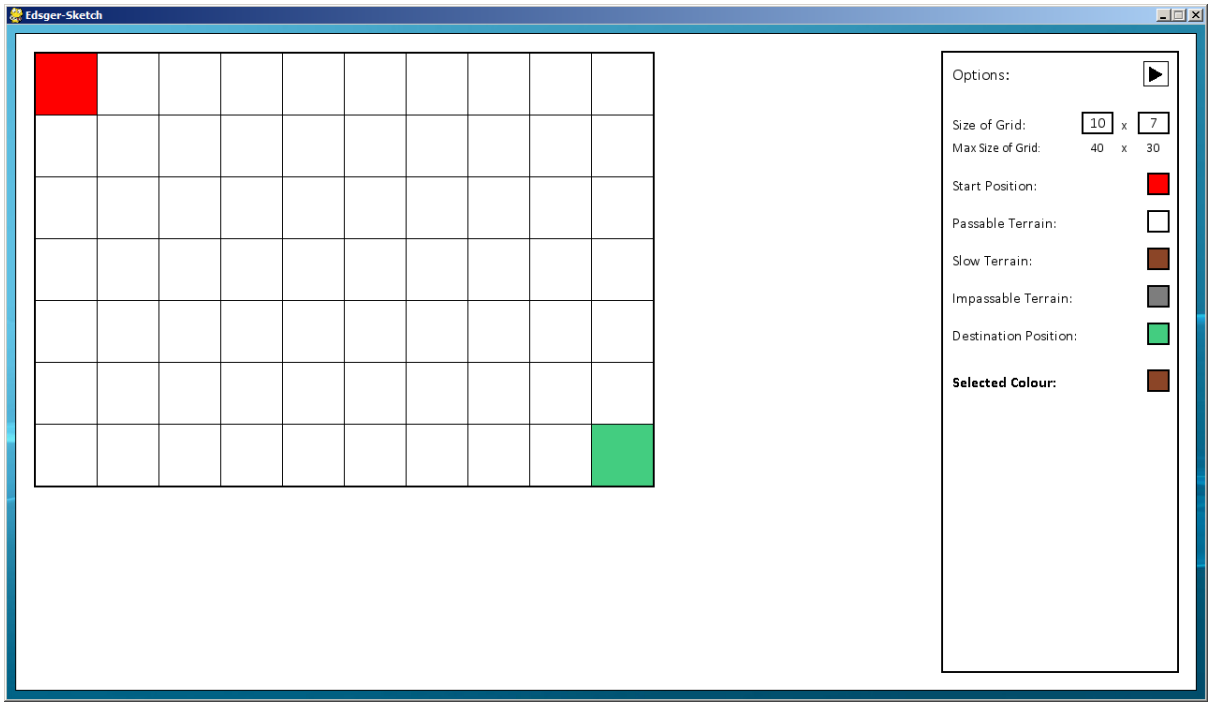


It uses the validation to change it to one cell and automatically updates the grid accordingly.

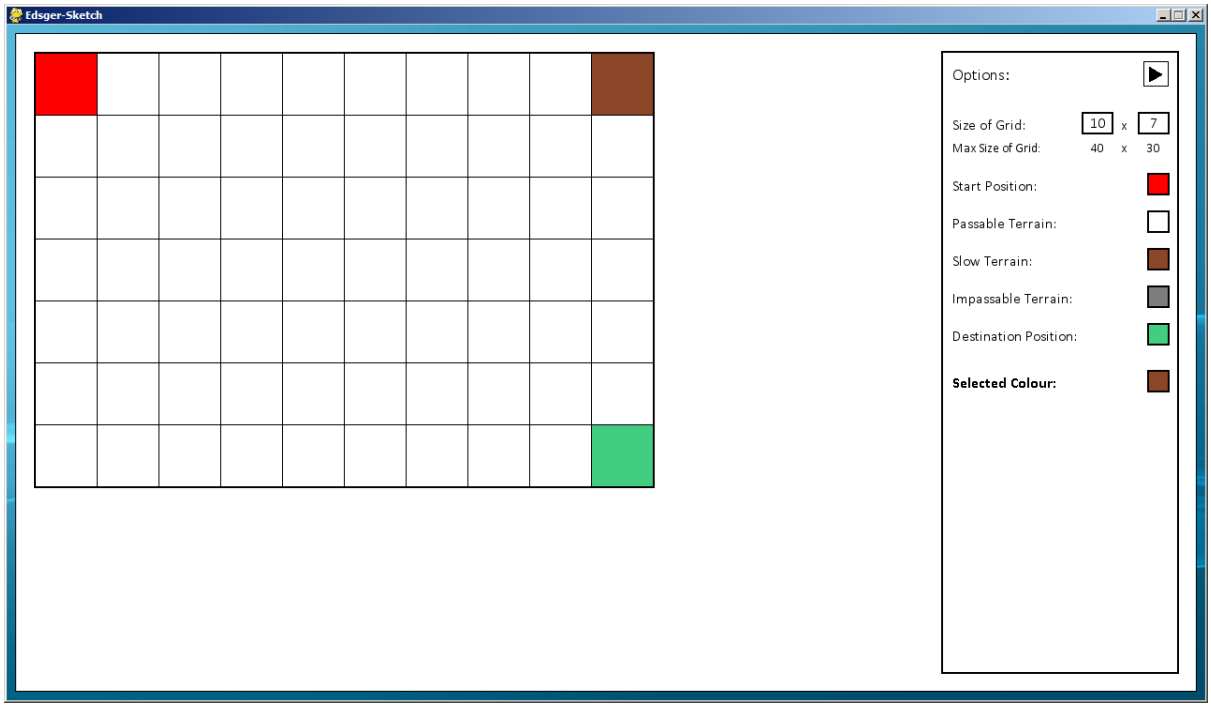
Test Set 4

Tested Data	Test Type	Expected Result	Achieved Result
Testing that the correct cells update appropriately when assigned a property/ colour	Normal - Assigning a cell with a normal cell property	The grid should update with the cells being an appropriate size	Expected result is achieved
	Extreme - Clicking the border of a cell	The grid should not register that a cell has been clicked, since the border of a cell could be in between two or more cells	Expected result is achieved
	Erroneous - Clicking outside of the grid	The grid should not change	Expected result is achieved
Testing that only one start and one destination can be placed	Erroneous - Entering a second start/ destination cell	Entering a second start/ destination cell should erase the first	Expected result is achieved
Testing that on the Vertices function, there can be no adjacent usable cells	Erroneous - Assigning two adjacent cells passable terrain	When assigning a cell with a property which is passable, it removes any cells from around it	Expected result is achieved

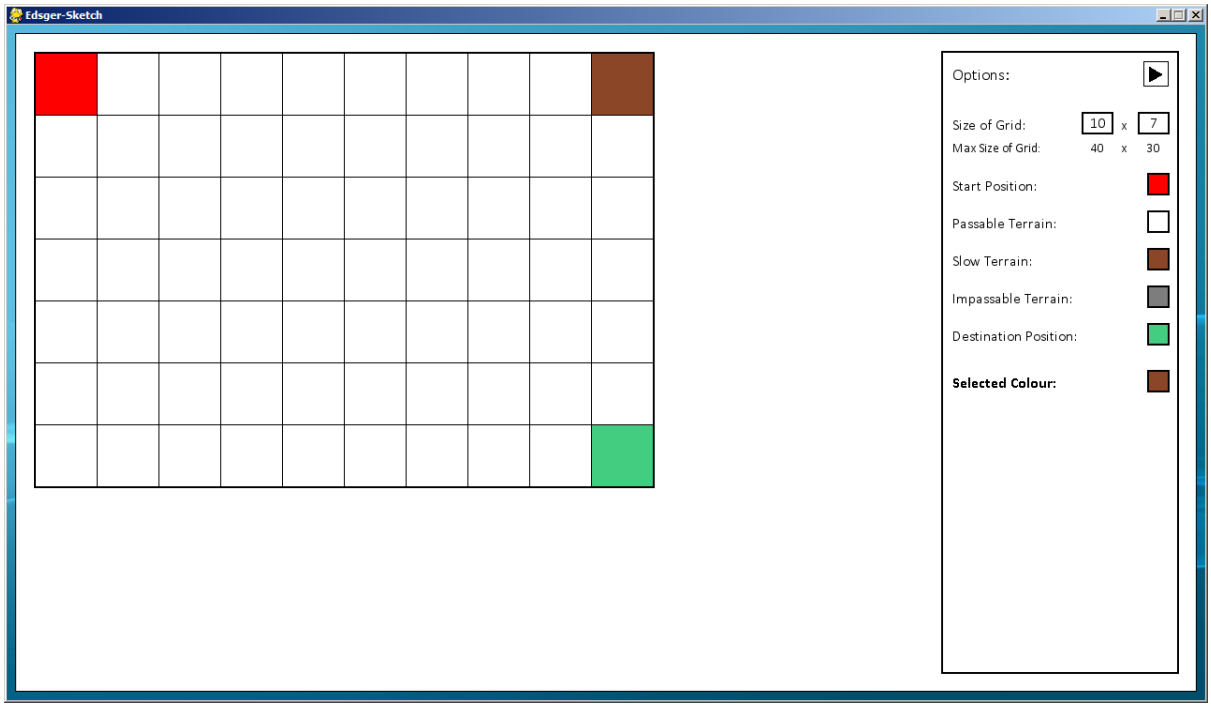
I will start by testing the Grid function:



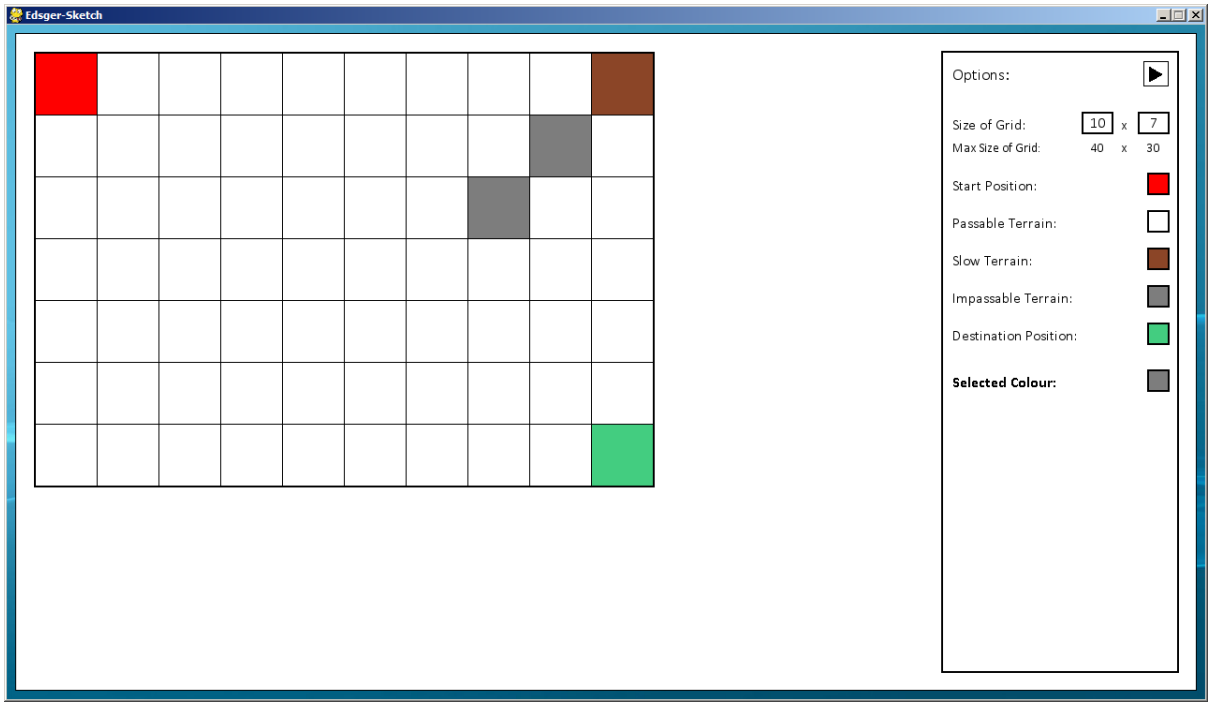
Here I assign a cell with the Slow Terrain property, this can be also be seen by the Selected Colour cell in the options, which shows the current property which will be assigned to a cell if a cell is clicked.



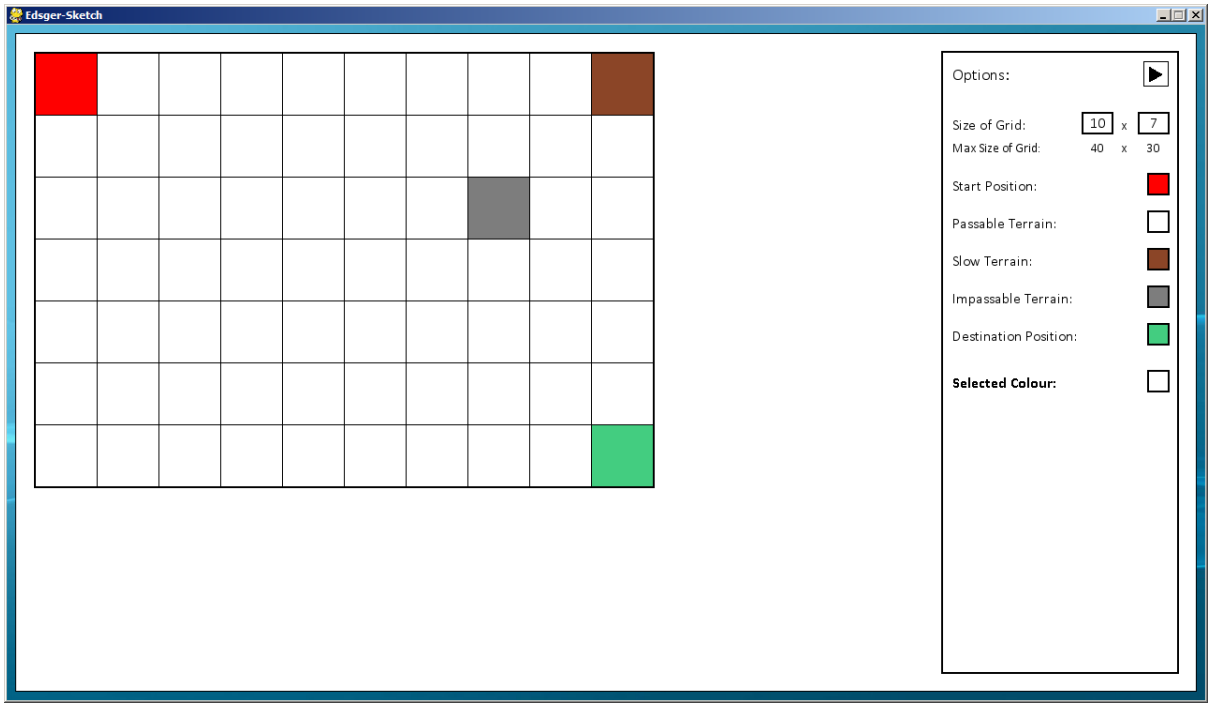
Clicking outside the grid, or on the border of a cell, results in nothing happening and the same screen being displayed:



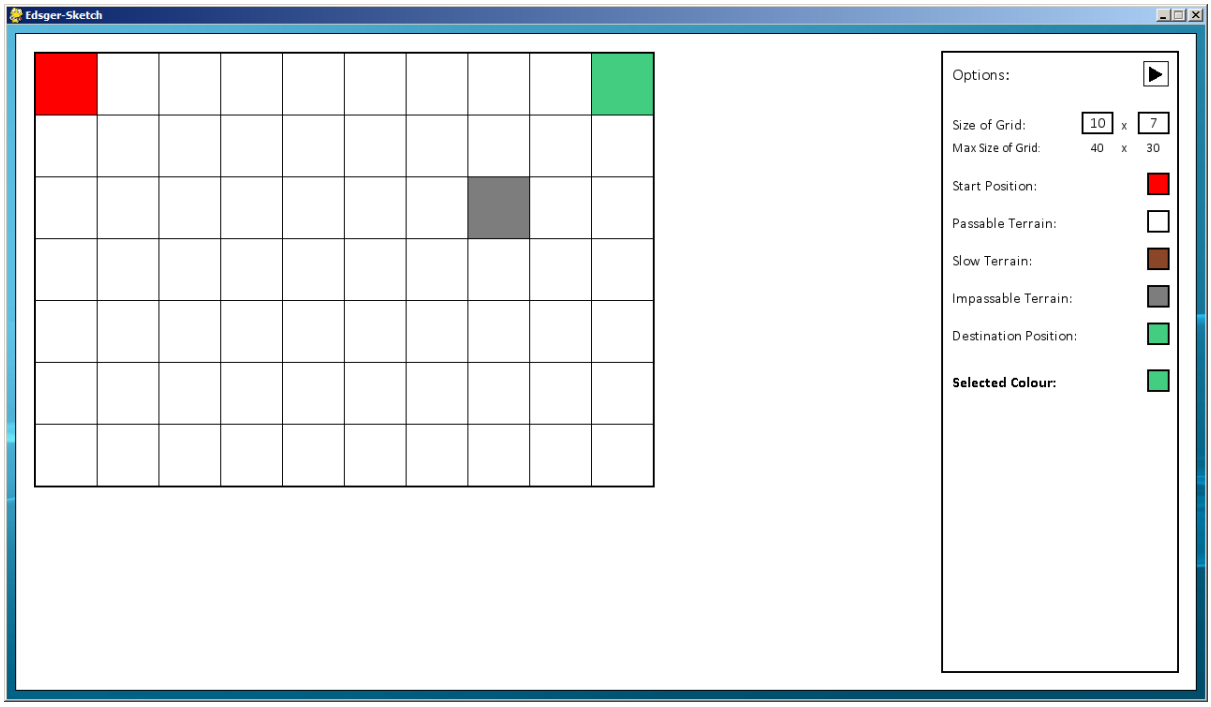
Here I test that the impassable terrain cell functions properly:



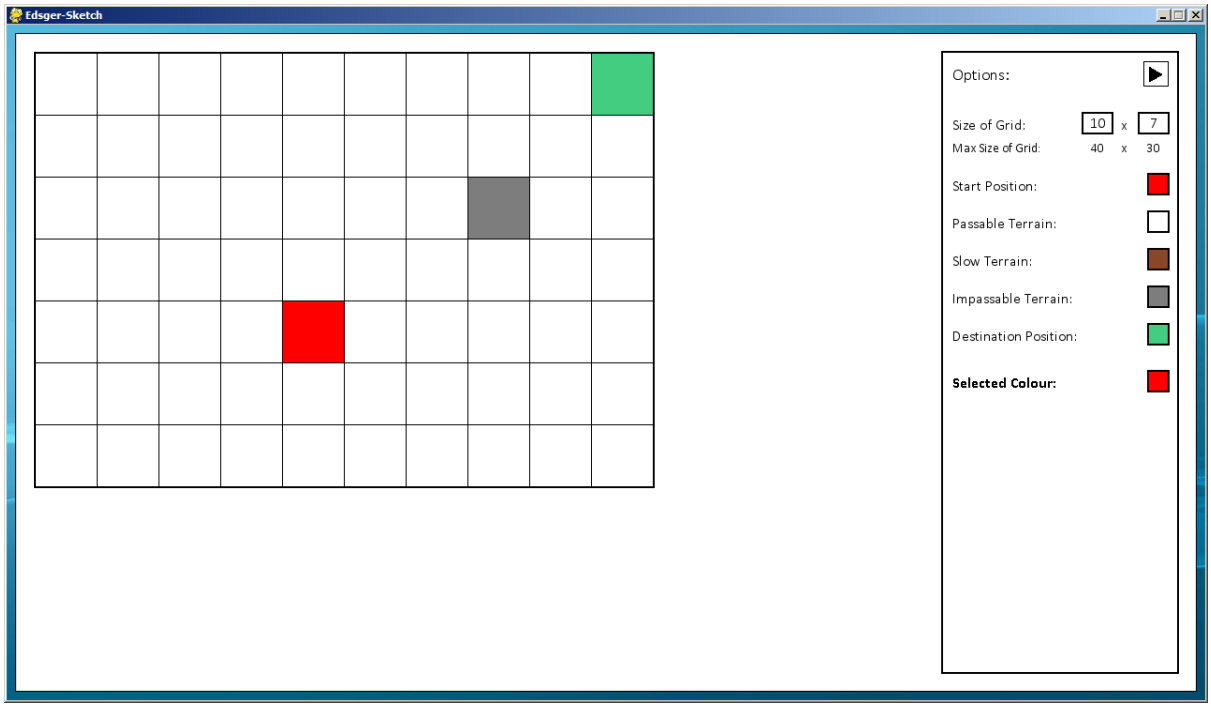
From here, since I have some cells with property, I can assign the cell the Passable Terrain property, to return it back to a normal cell:



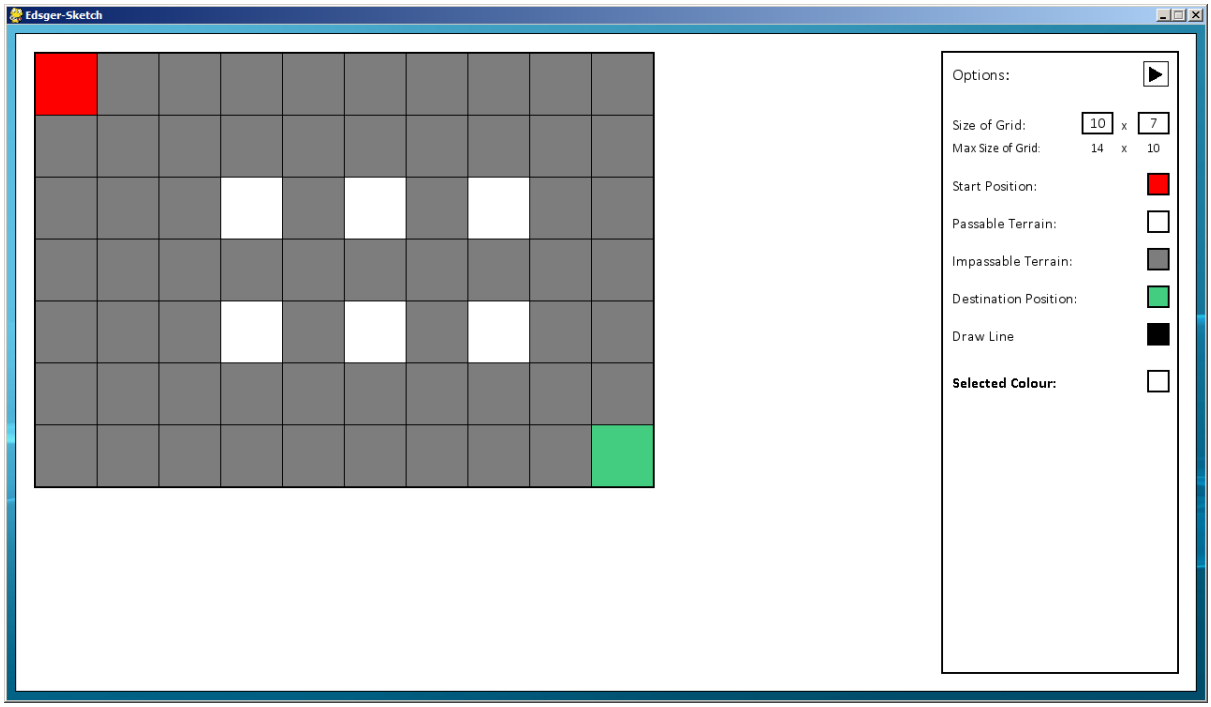
Here I test replacing a cell with the Destination Cell property, as you can see since I can only have one destination cell it replaces the old cell.



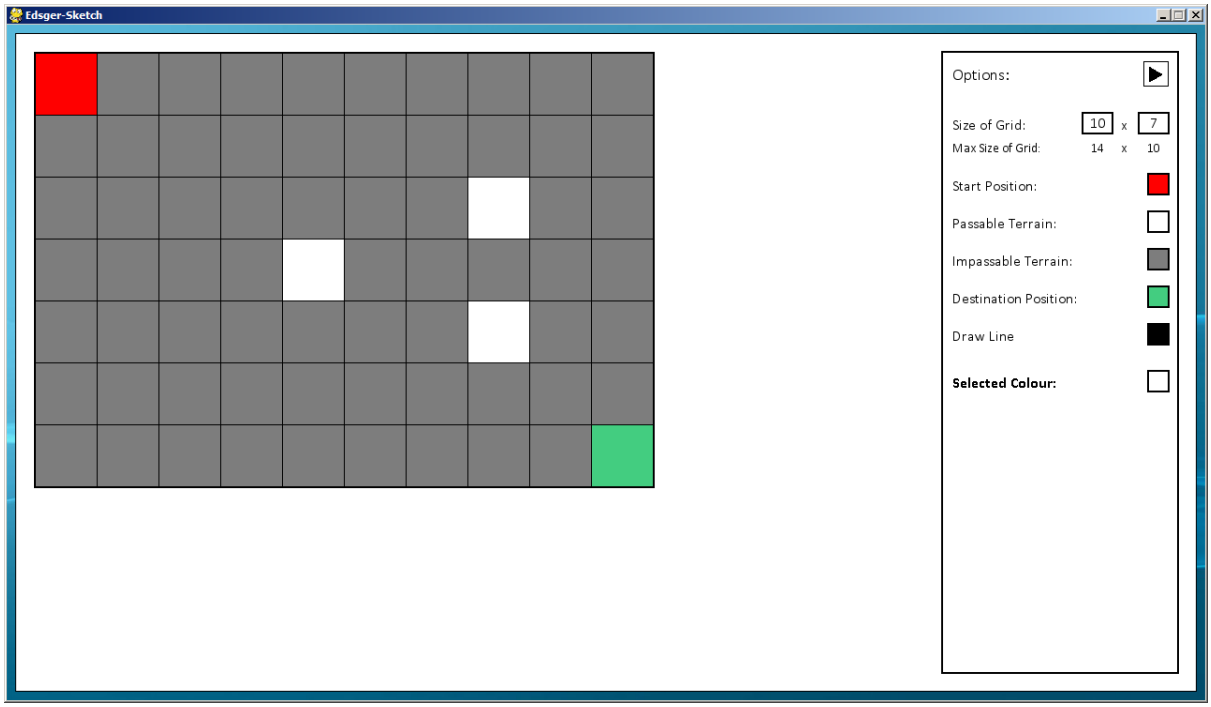
Here I do the same with the start cell and the same thing happens.



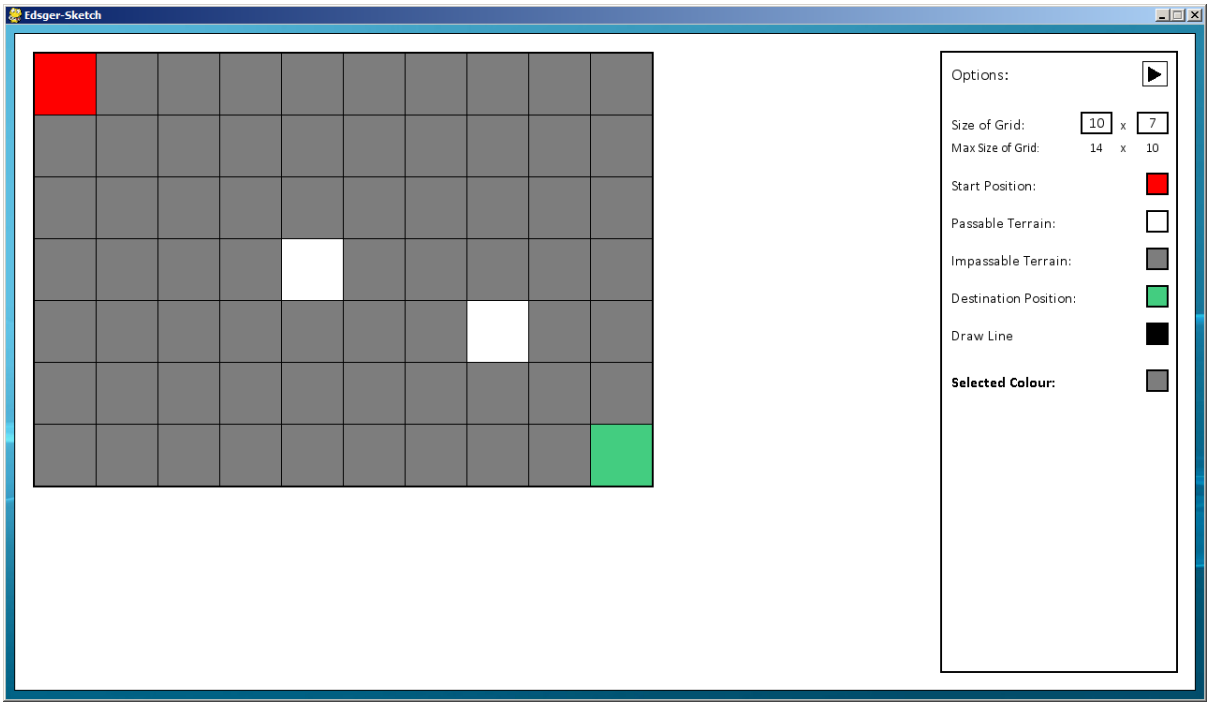
Here I move on to the Vertices function to test that a cell which can have a line drawn on to it, a property which is passable, cannot be next to another cell with this property. This is the screen I will be working with:



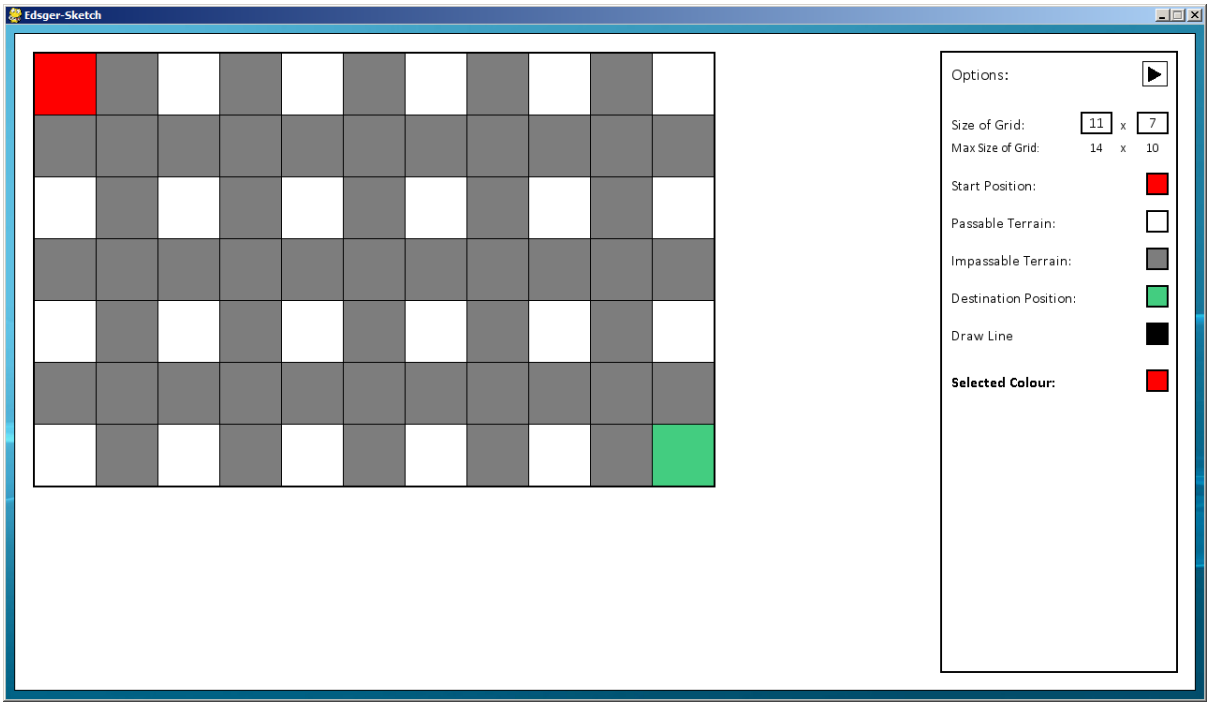
I then assign the cell in the middle of the 4 passable terrain cells on the left, with the same property; since they cannot be adjacent it removes the others from the grid:



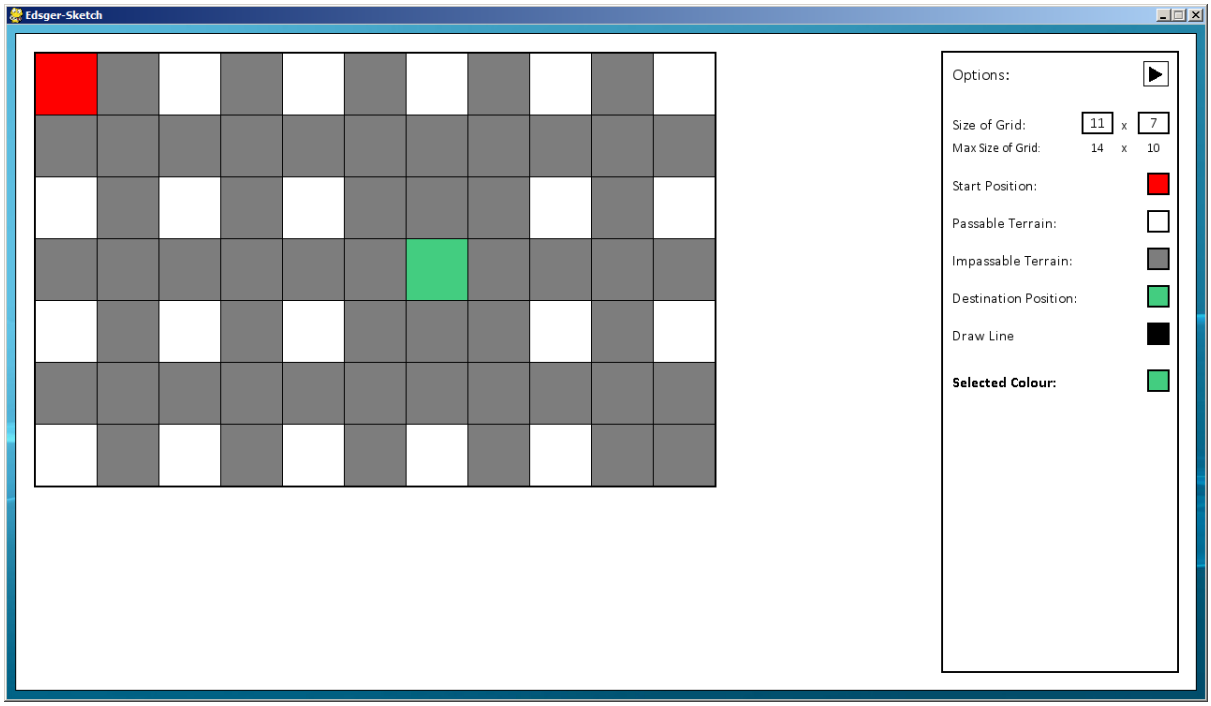
I then try assigning the top right cell with the impassable terrain property, which means it cannot be used to connect lines to anymore.



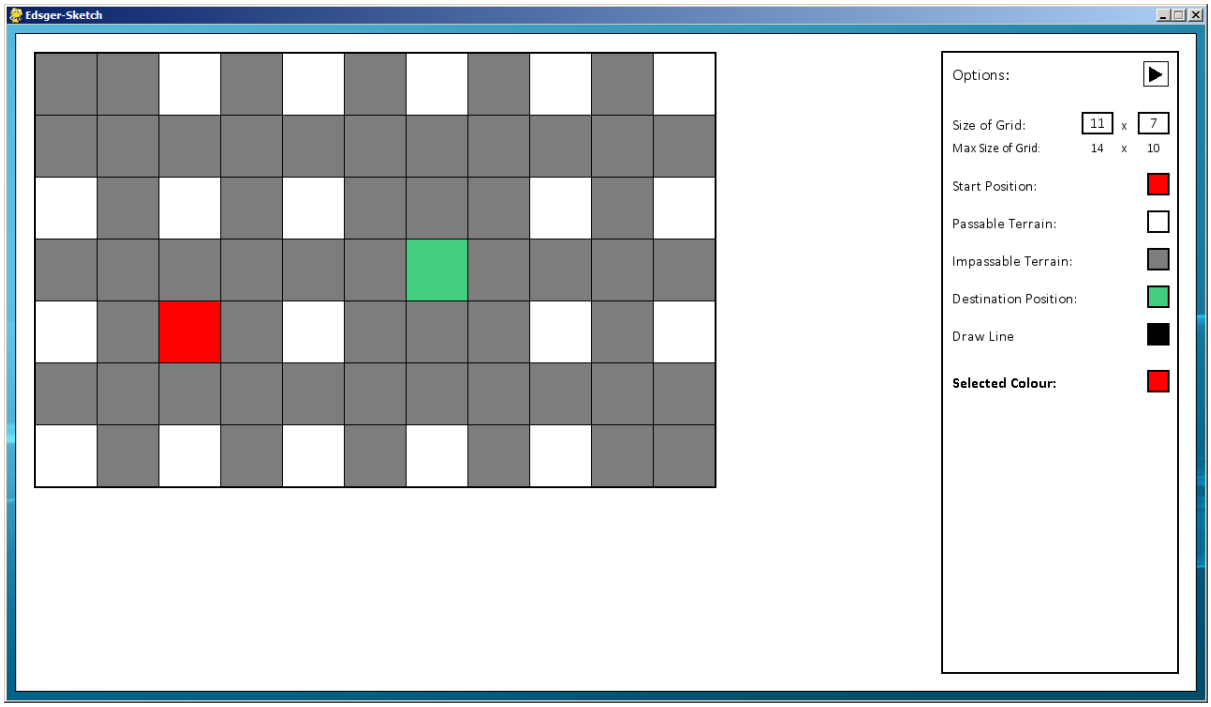
Here I start with a new blank grid to show it only removes those cells that are adjacent to it:



Here I replace a cell in between only two cells.



Here I simply replace the cell with another property, so it doesn't change any cells around it.



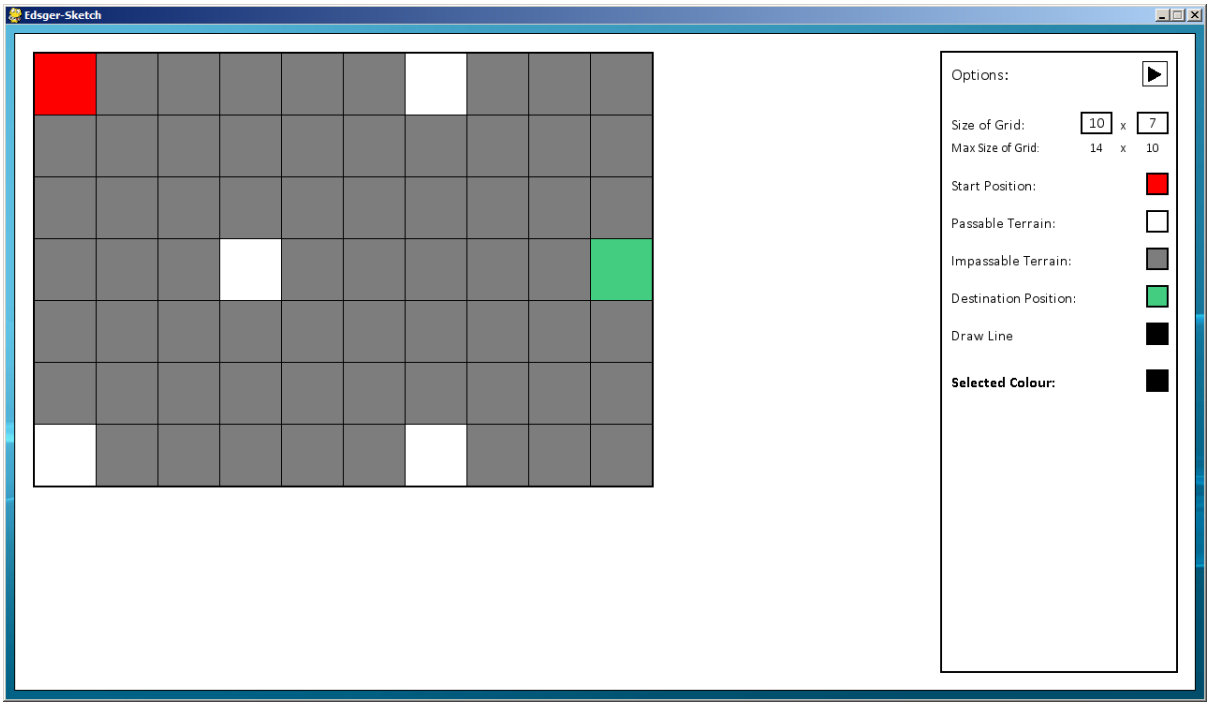
Note that again, the original start and destination cells are removed from the grid.

Test Set 5

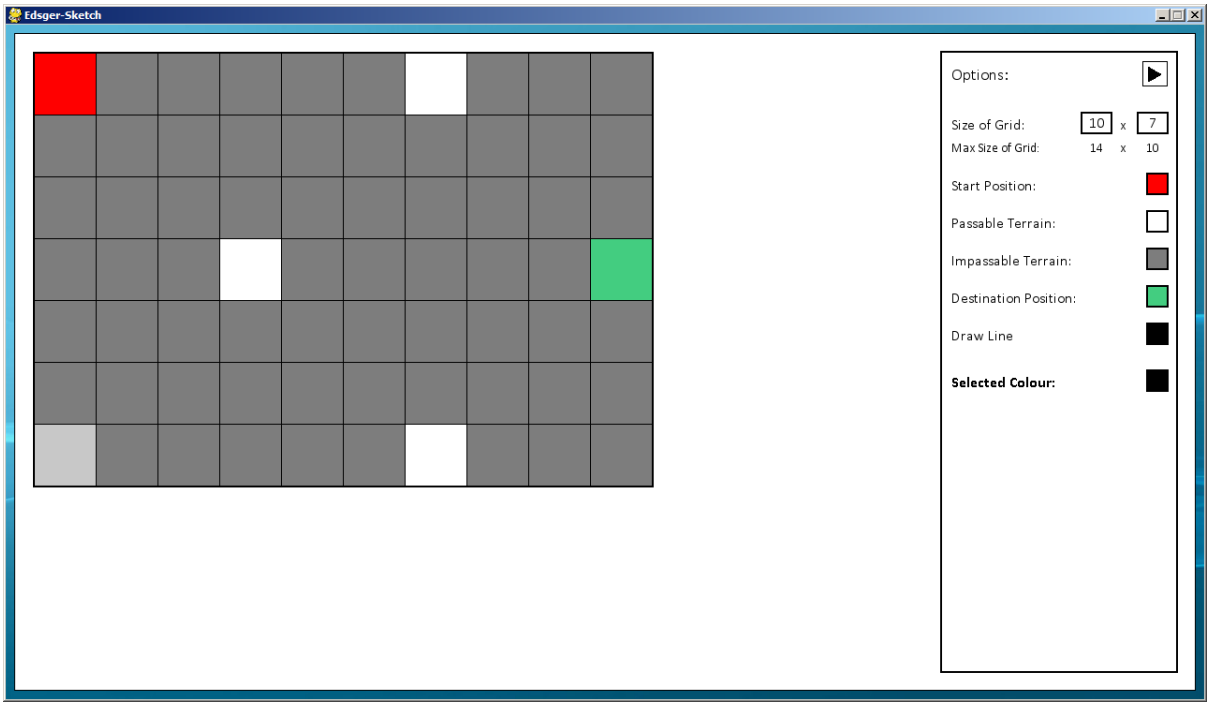
Initial Tests

Tested Data	Test Type	Expected Result	Achieved Result
Adding lines to the grid	Normal - Clicking from one passable cell to another	First cell should be coloured light grey. Upon clicking a second cell a line should be drawn with the distance to two significant figures displayed in the middle	Expected result is achieved
	Erroneous - Clicking from a passable cell to an unpassable cell or outside the grid	Line should not be drawn. If the second click is outside the grid the starting cell is removed	Expected result is achieved
	Erroneous - Drawing a line over another line/ intersects with another line	The line should not be drawn and the starting cell removed	Expected result is achieved
Editing cells with lines drawn to them	Erroneous - After a line is drawn between two cells, try and edit the properties of one of the cells	Options to edit cells should be disabled until the lines are cleared from the grid	Cells can still be edited, resulting in lines starting/ ending on impassable cells
Clearing lines drawn	Normal - Clicking the button	Lines should be removed from the screen and associated variables cleared for further input	Expected result is achieved

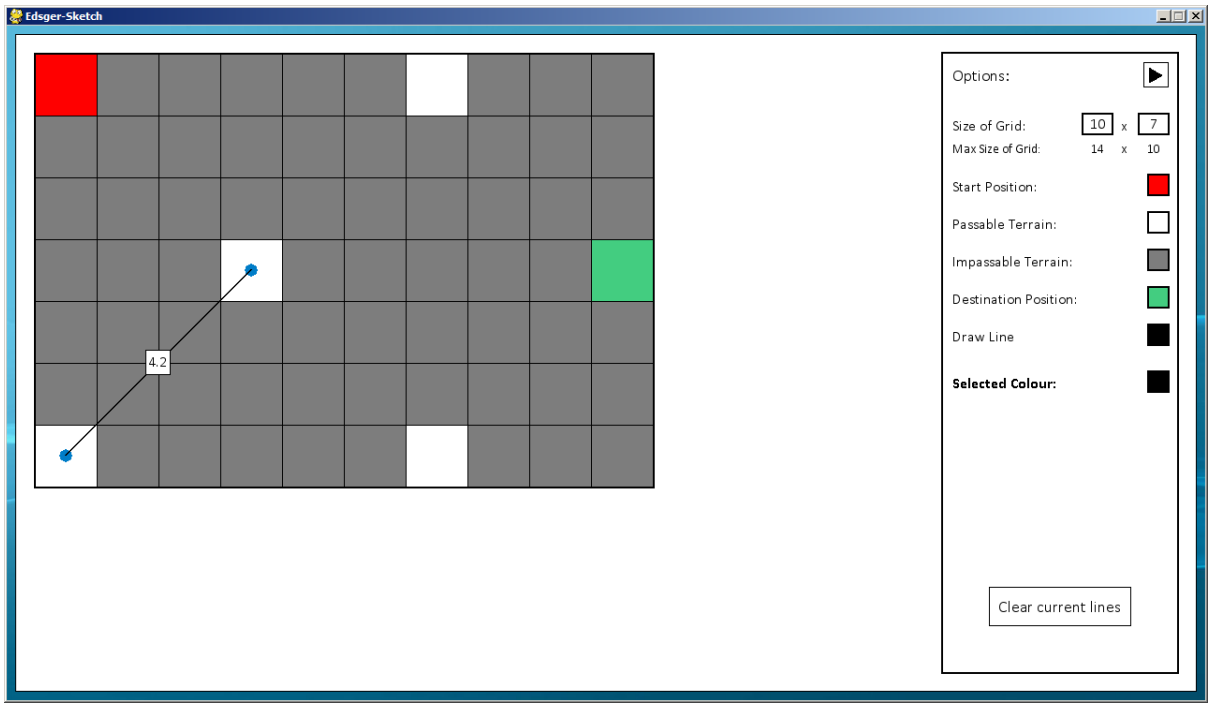
Here is the Vertices function screen that I will be testing, since lines can't be drawn on the Grid function I will not be testing that function.



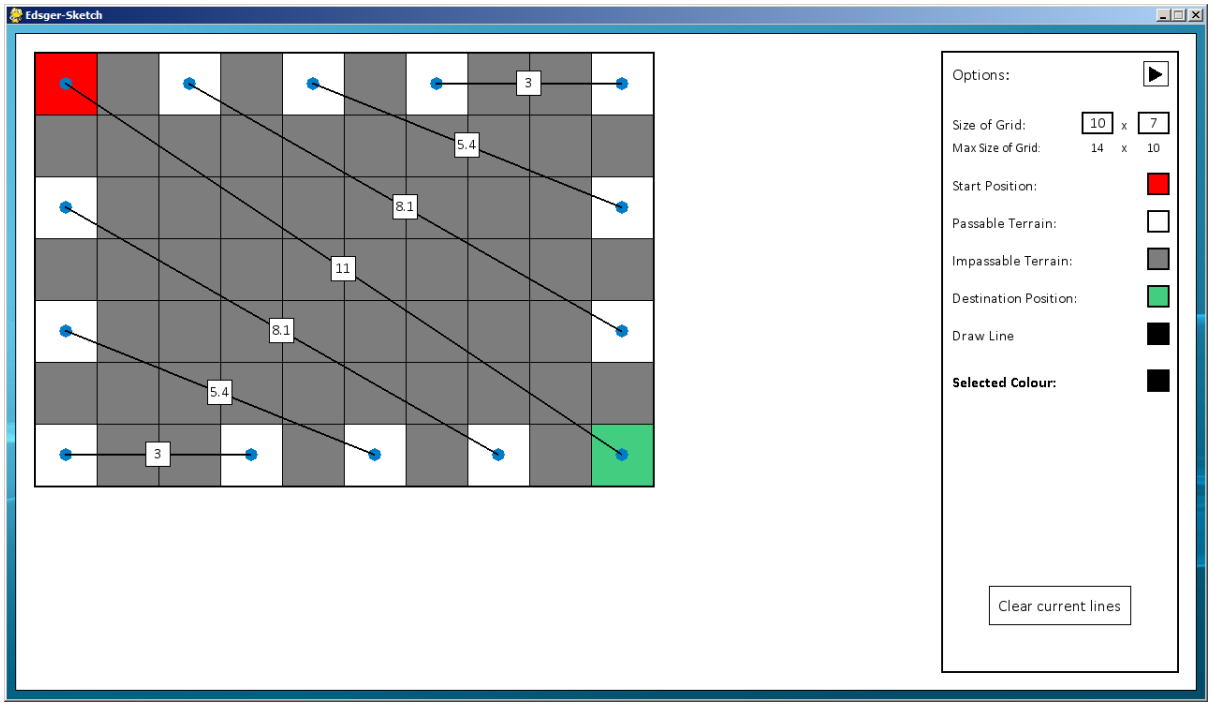
Here is test that the first cell to be selected when drawing a line is shaded a light grey, so the user knows where the line will be drawn from.



Here the line is drawn, with the length of the line, to two significant figures, shown in the middle of the line.

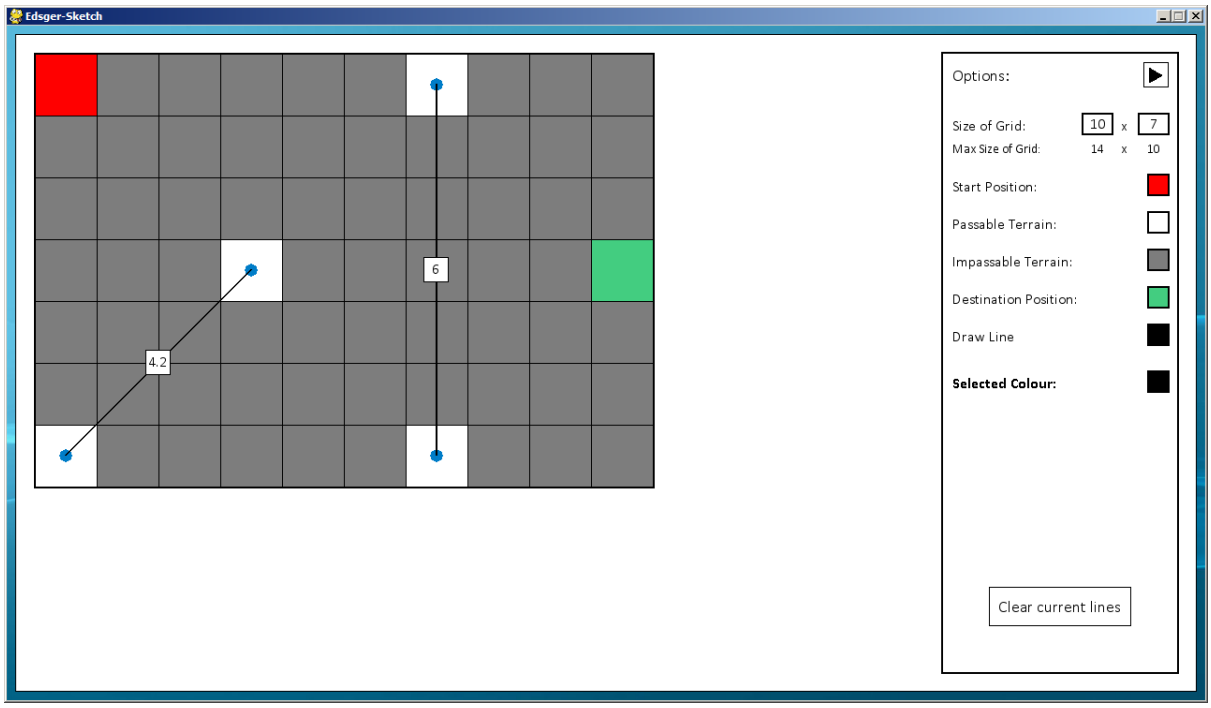


Here is another example, showing that when the length is longer than 10 cells, it no longer shows the decimal place.

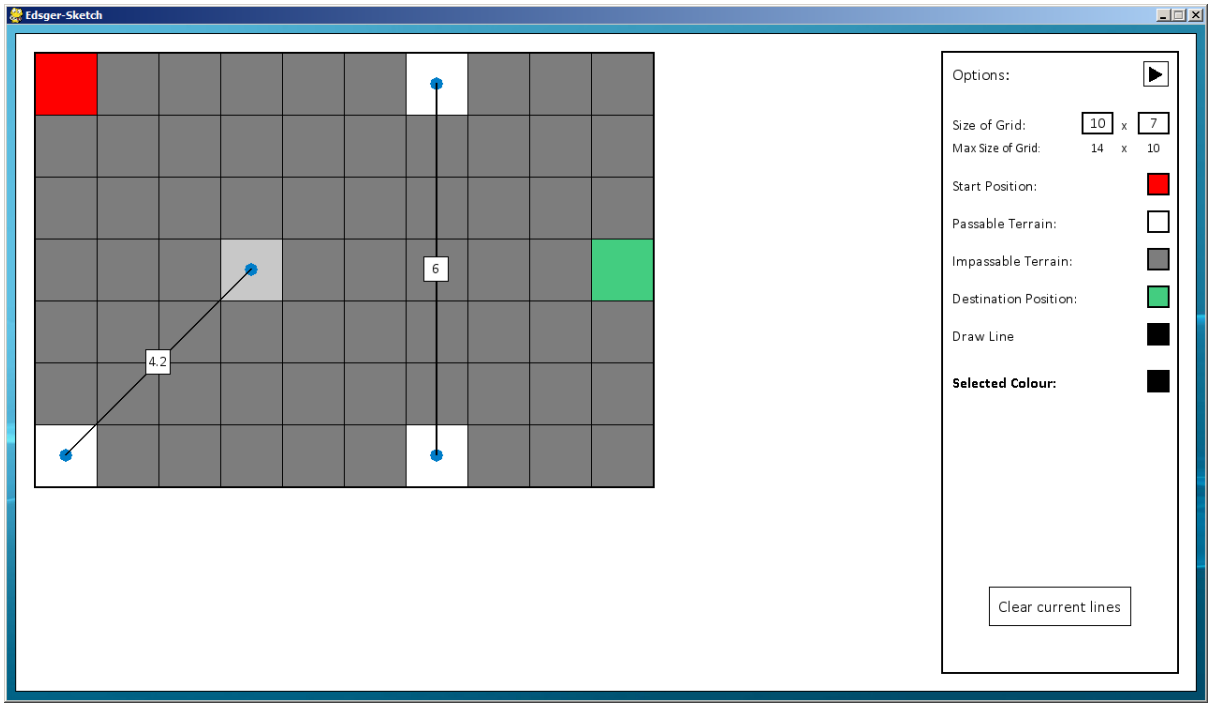


As you can see the line in the middle has a value of 11, rounded from 10.8.

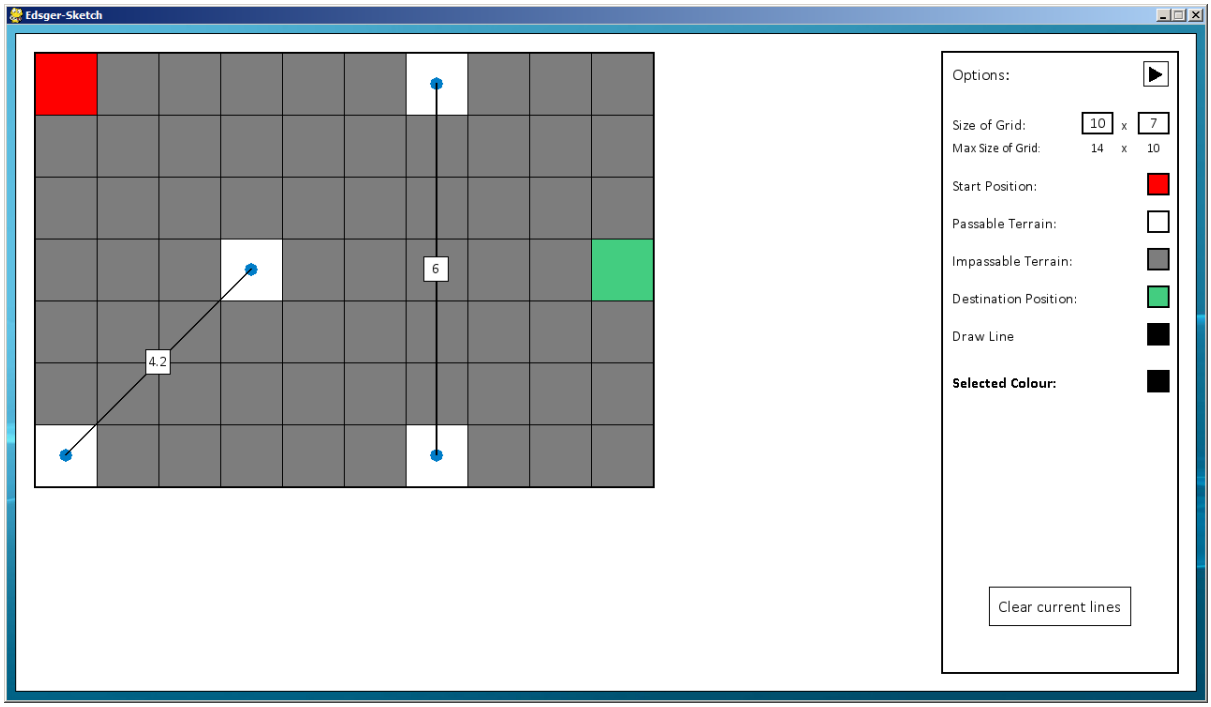
Here I am going to test what happens when trying to draw a line that intersects with another line:



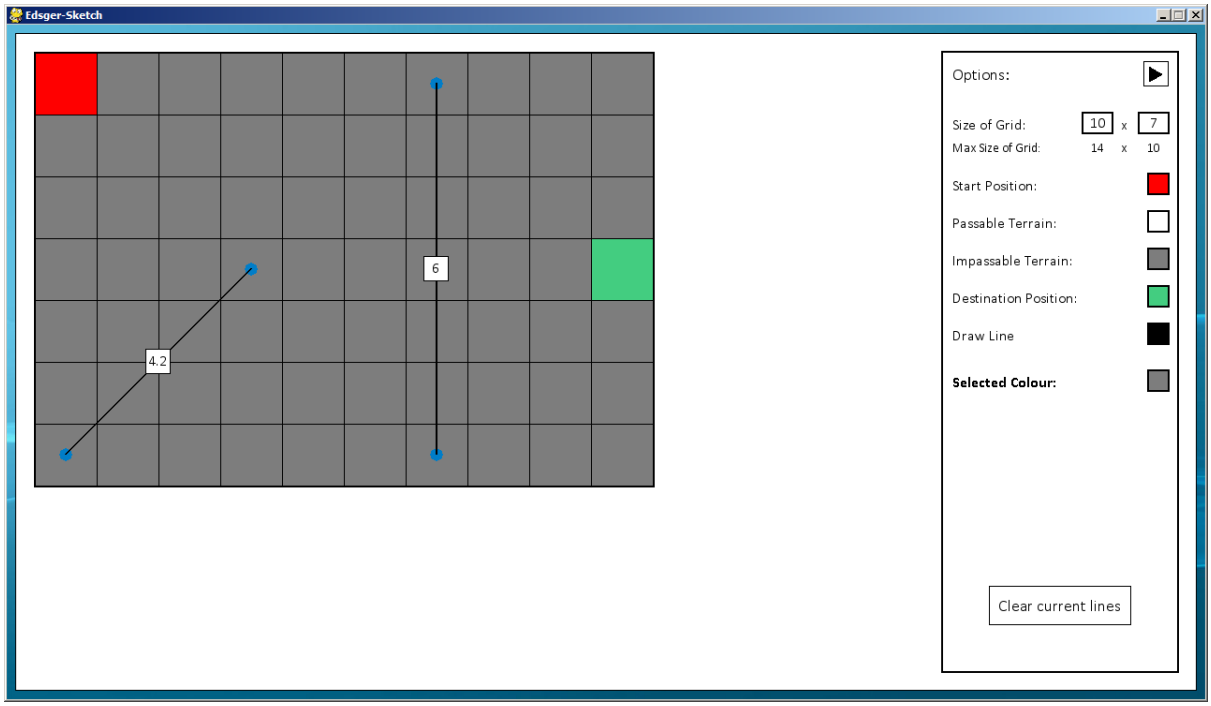
Here I select the cell to the left of the vertical line.



When trying to draw a line across, since it is invalid it does not draw the line and resets the starting cell for the line to be drawn from. This validation is in place so that the screen doesn't become too confusing with lines intersecting everywhere.

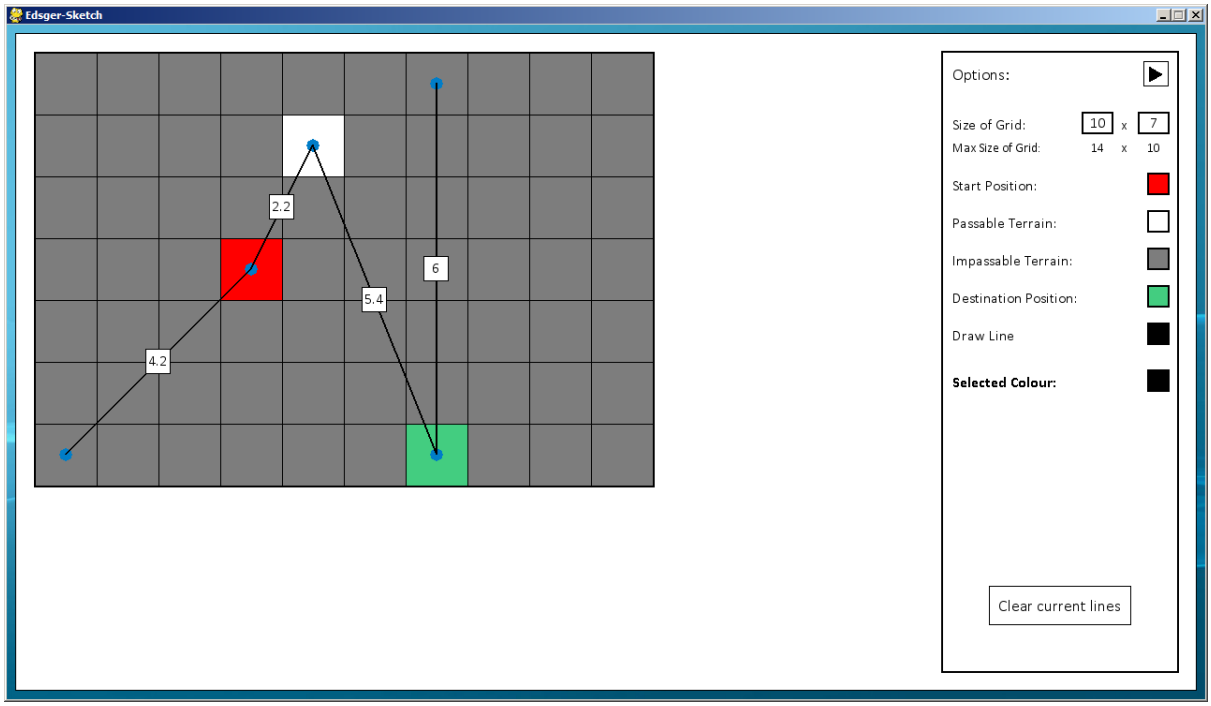


I now test editing cells with lines drawn.

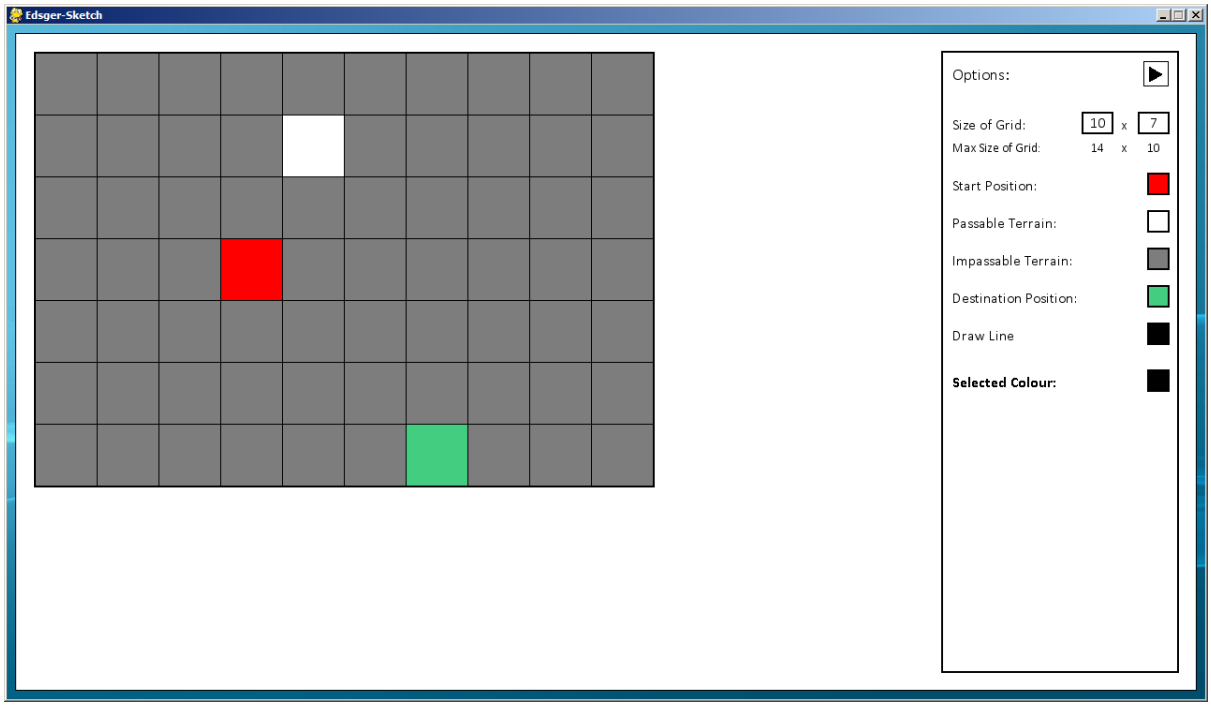


As you can see I have found a fault with my program, it allows me to edit cells with lines attached to them, meaning that lines are not connected to cells.

Here is another test showing the flaw with my program.

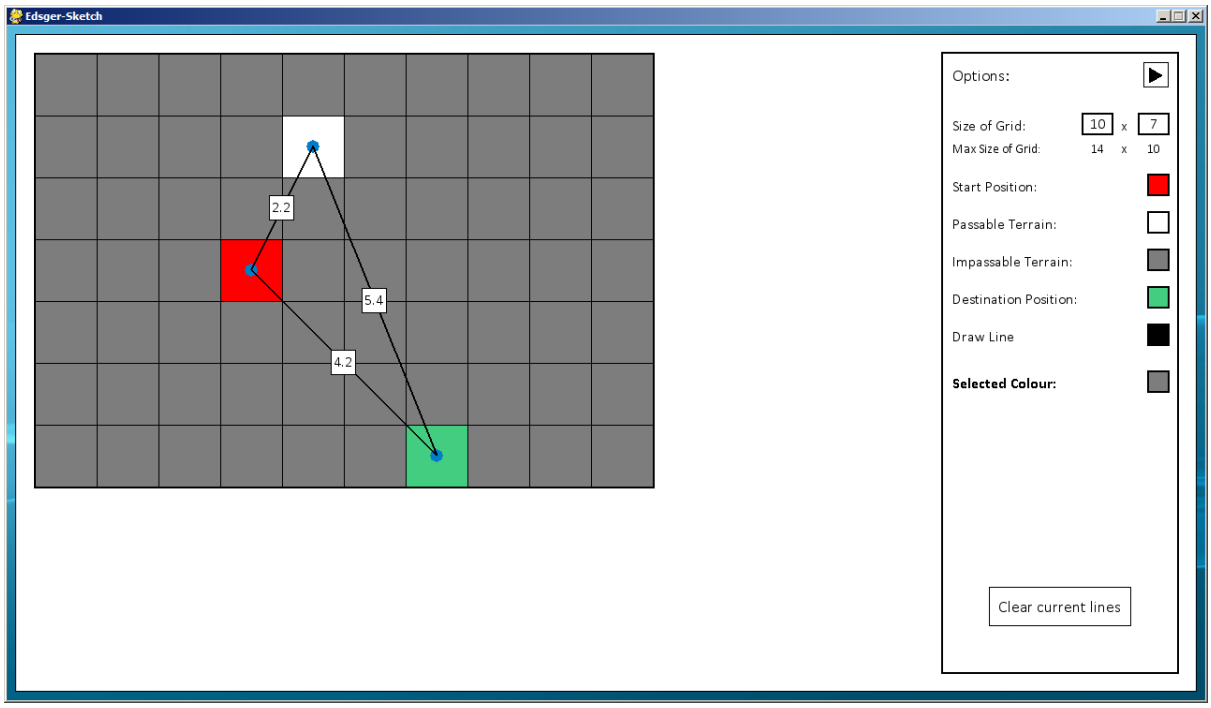


Here I am testing that the "Clear current lines" button functions correctly.



Fortunately this still works fine despite the other faults with my program, as can be seen it removes all current lines that have been drawn.

Here I test that I can still draw lines when the old lines have been cleared without any faults.

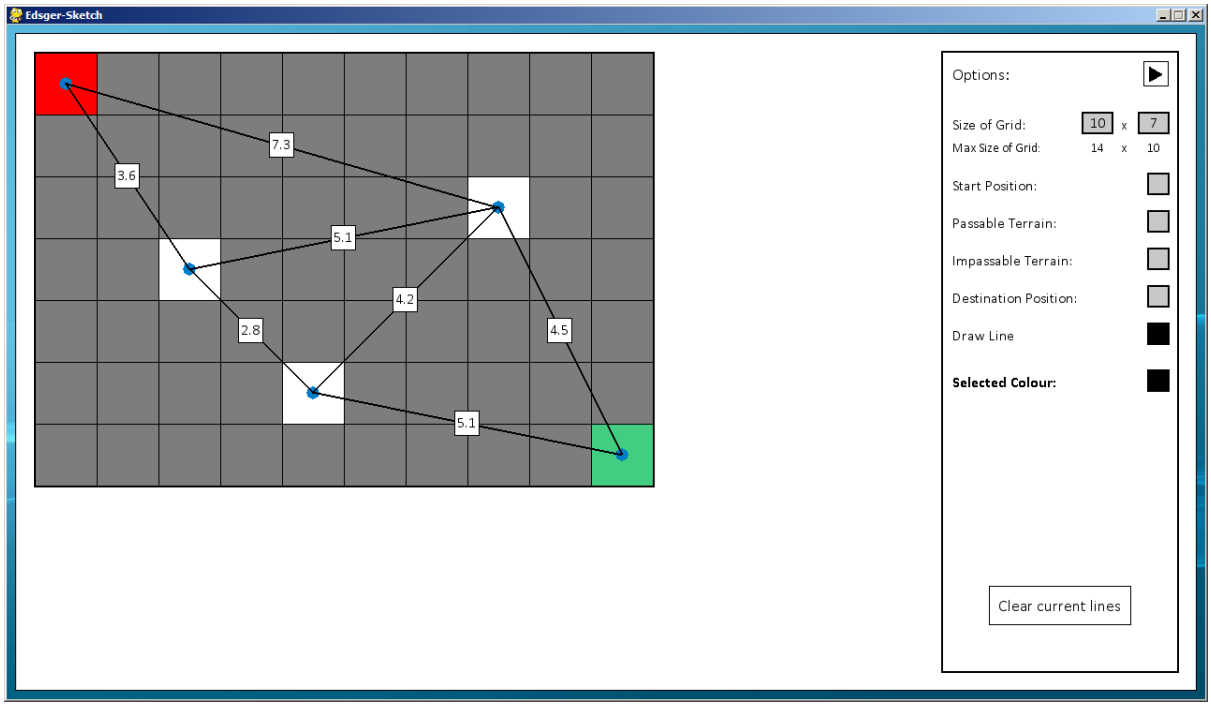


Fixing Faults

The fault with my program was a simple error on my behalf. I had forgotten to add validation to this section of the program so the user could easily break the program, after adding appropriate validation in to the program, I then conducted further tests to see if this new validation worked.

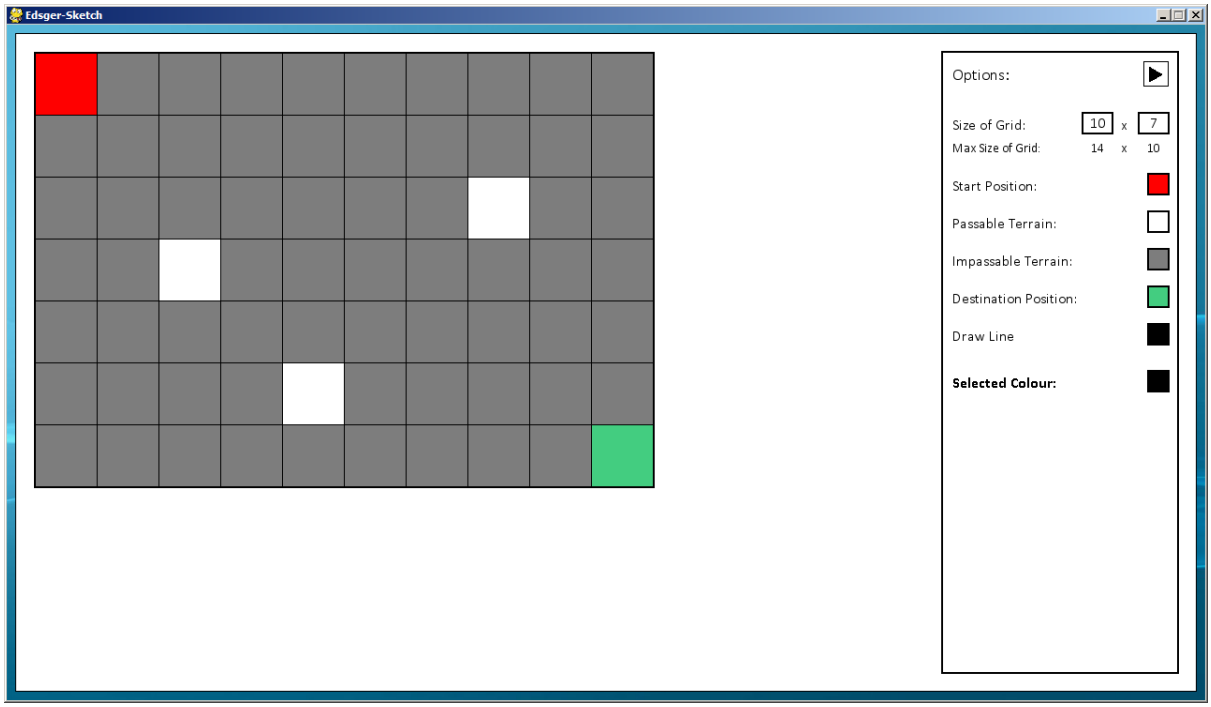
Further Tests

After fixing the fault with my program I have carried out some further tests. Here I show that now, when lines have been drawn on the screen, you can no longer edit the cells and they are greyed out.

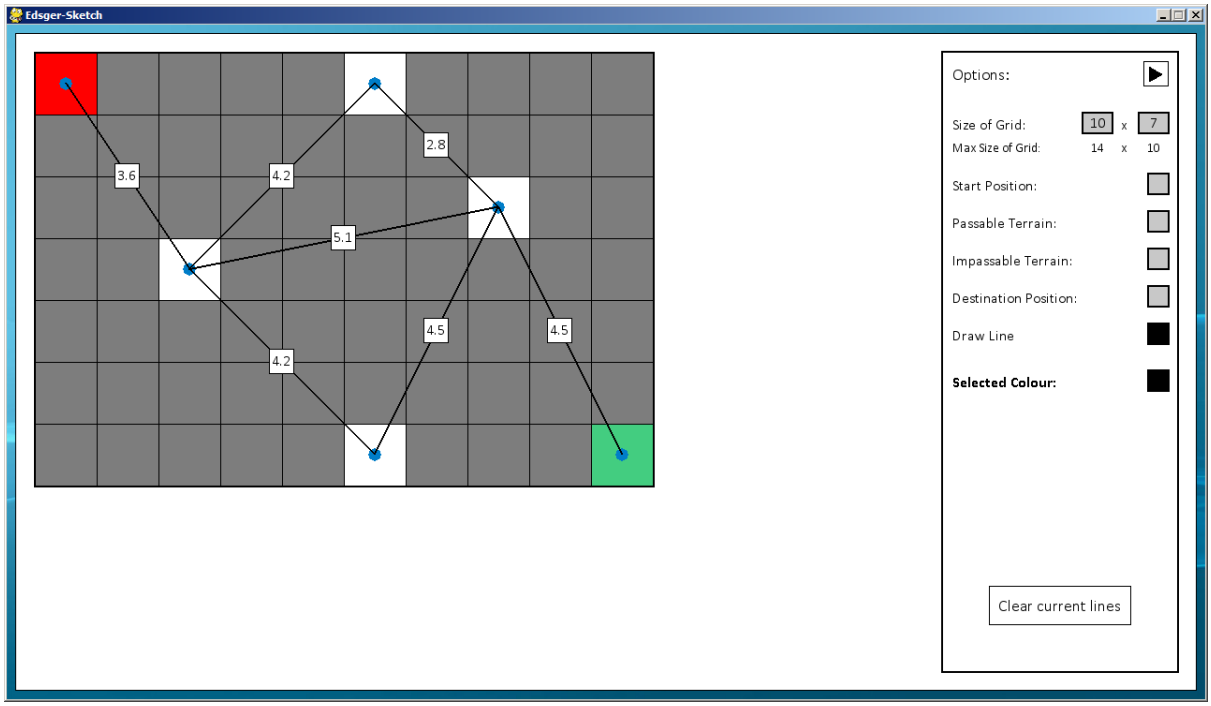


When trying to select a different property nothing registers.

I now test that the clearing lines button functions correctly still.



Here I test that when clearing the lines, I can continue to edit the grid.



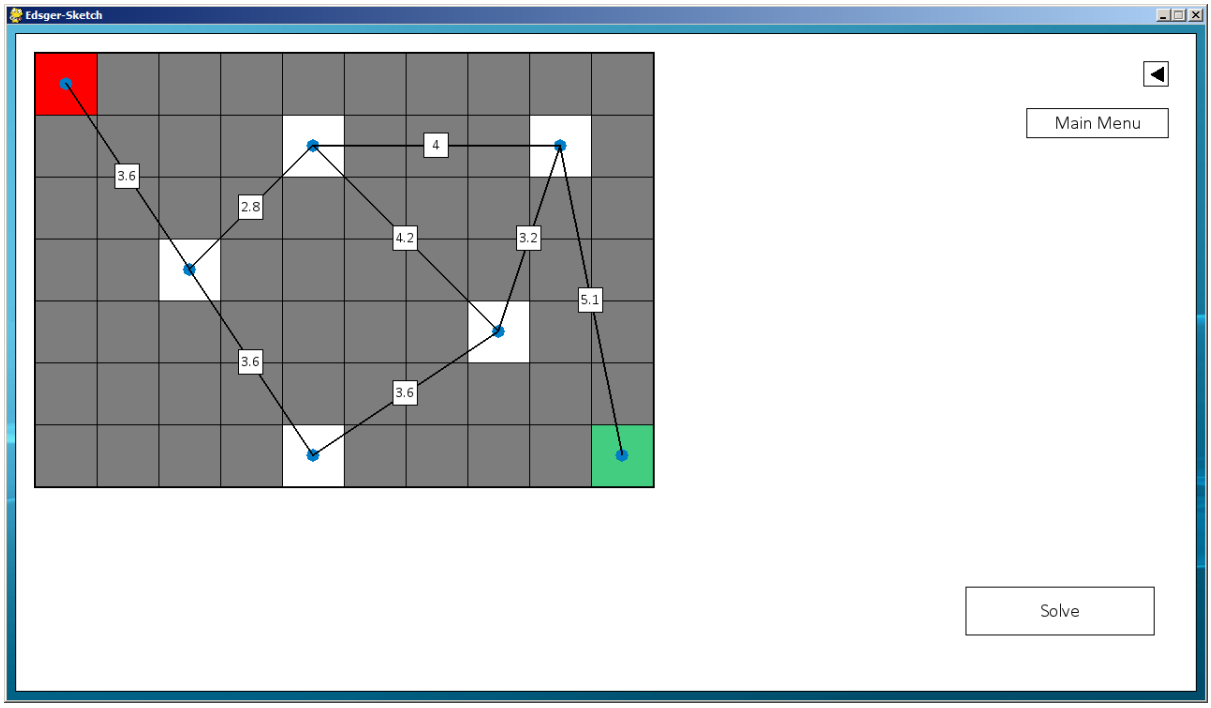
Test Set 6

Initial Tests

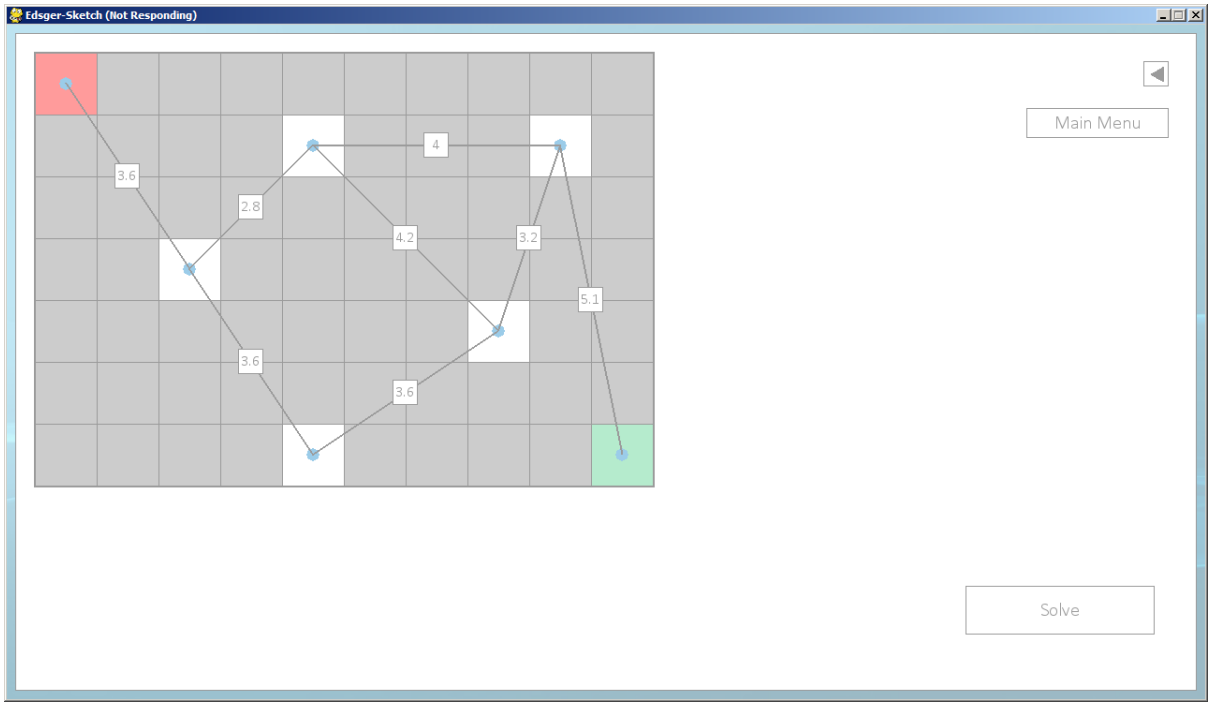
Tested Data	Test Type	Expected Result	Achieved Result
Solving the graph	Normal - Solving a graph with lines connecting the start and destination cells	The lines forming the shortest route from the start cell to the destination cell should be highlighted, showing the shortest route	The program stops working once the Solve button has been pressed

Since this is such a significant problem with my program, I need to fix it before I can conduct further tests.

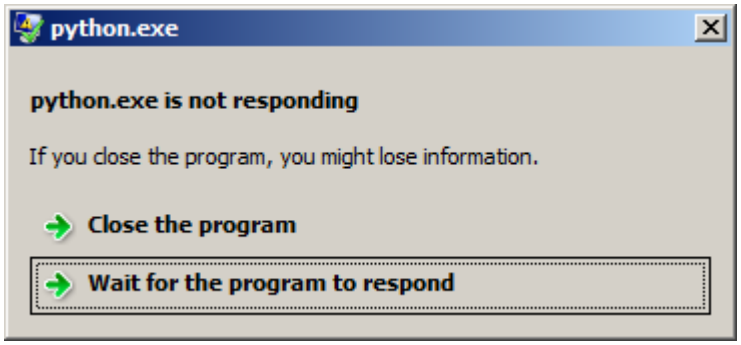
Here is the screen that I tested.



When clicking "Solve", the program becomes unresponsive.



Eventually the program crashes, resulting in this error message:



Fixing Faults

Initially, to find what part of the program was causing the problem, I used the bottom-up testing methods. I tested each of the individual functions, by printing the inputs and outputs from that function. Eventually I found that it was a problem with the 'function2' function, within the 'perform_dijkstras' file. I then employed the white box testing method to find what was wrong with the program; using pen and paper I drew the grid by hand and what the result should be. I then tested the outputs I was getting against my own results, when given appropriate data. Eventually I found that the program worked fine without floats, however I was encountering a problem with floating point arithmetic. Here is an example of part of the code I used to identify the problem; it is a modified extract from part of the function.

```
final_route = []
current_cell = [10, 4, 16, 75.4, 75.4]
i = [10, 4, 8, 6, 2.8]
j = [8, 6, 15, 72.6, 72.6]

if i[2:4] != current_cell[:2] and i[2:4] == j[:2]:
    a = current_cell[3] - j[3]
    b = i[4]
    print a, b
    if a == b:
        print "True"
        final_route.append(i)

print final_route
```

The result being printed on to the console was:

[print a, b]: 2.8 2.8

[print "True"]:

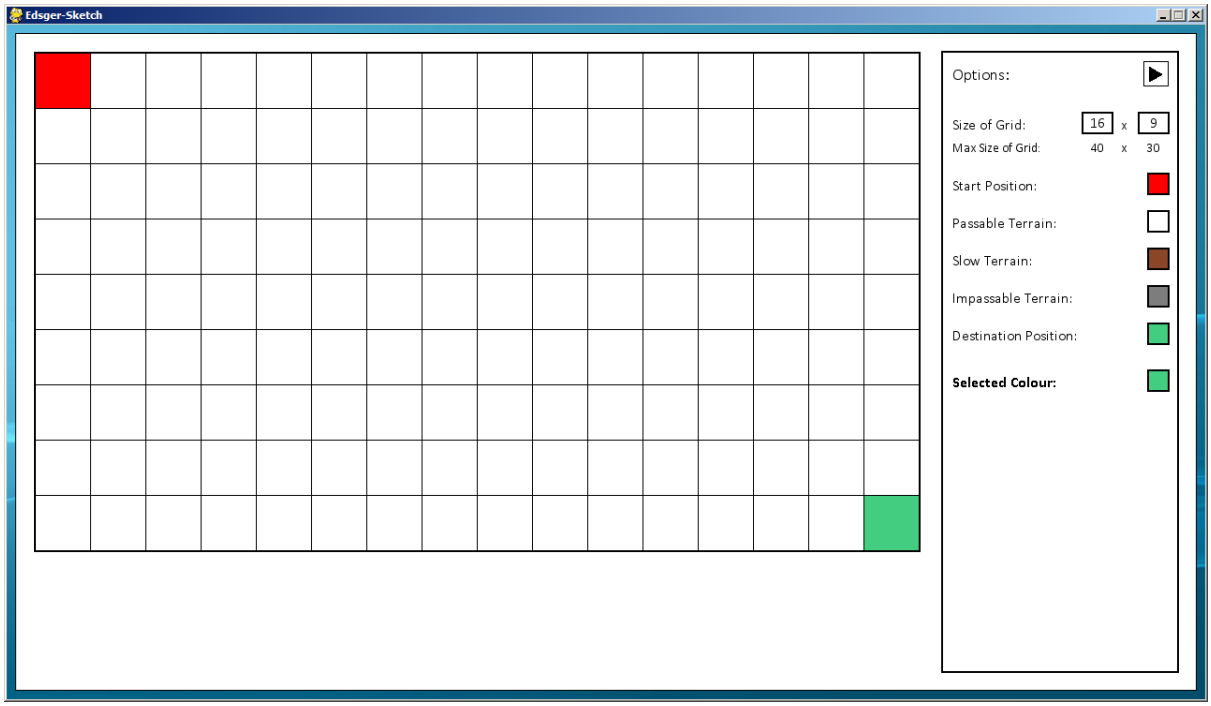
[print final_route]: []

Here, even though a and b were the same number, it wasn't accepting that they were and nothing was being appended to the final_route list, resulting in an infinite loop. After extensive researching I found an article [1] that identified the problem I was encountering, basically since floats can't be accurately represented using binary, so the value for a was actually 2.8000000000000114. To fix this I decided to times all the number by 10, since the lengths of the lines only ever went to one decimal place, then divided the lengths by 10 again at the end of the function when returning the final route.

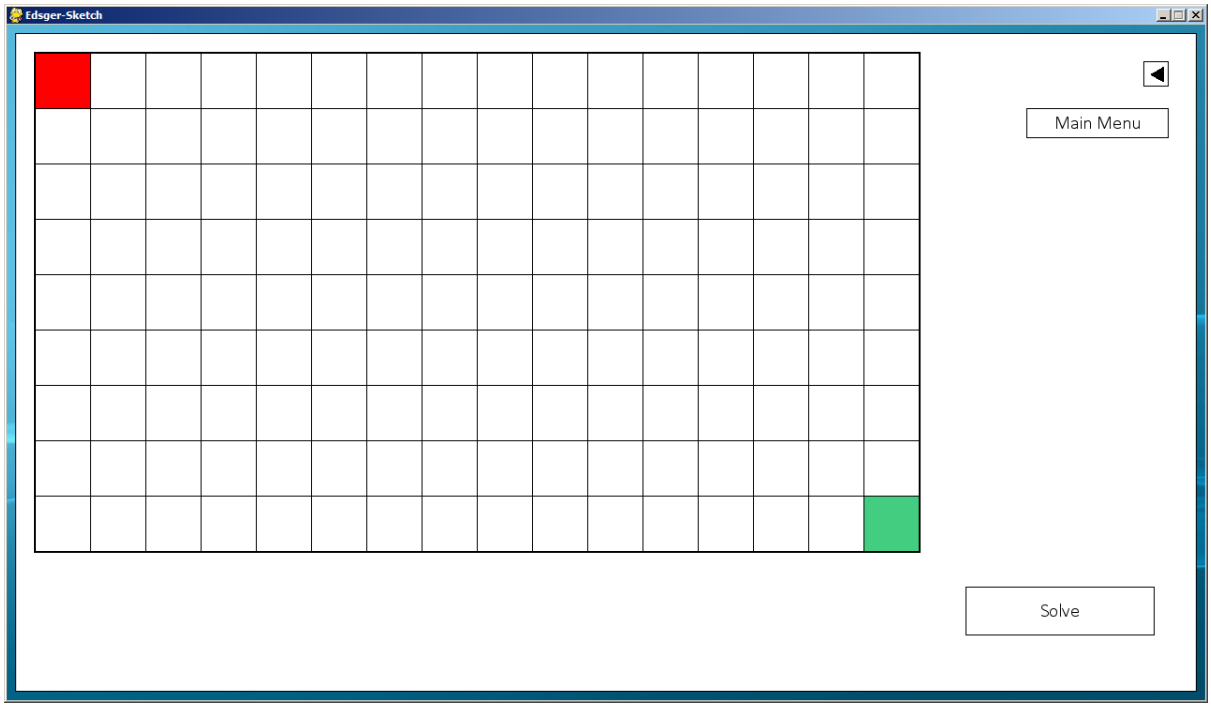
Further Tests

Tested Data	Test Type	Expected Result	Achieved Result
Solving the graph - Grid function	Normal - Solving a grid with both the start and destination cell present	A line should be drawn shown one of the shortest routes available, if there's more than one route available	Expected result is achieved
	Erroneous - Solving a graph without the start or destination cell present	The solve button should not be shown so the user cannot solve the grid	Expected result is achieved
	Erroneous - Solving a graph without a route from the start cell to the destination cell	When the solve button is clicked no line is drawn and the return button is shown	Expected result is achieved
Solving the graph - Vertices function	Normal - Solving a graph with lines connecting the start and destination cells	The lines forming the shortest route from the start cell to the destination cell should be highlighted, showing the shortest route	Expected result is achieved
	Erroneous - Solving a grid without the start or destination cells connected to lines	Solve button shouldn't be shown as a basic level of validation	Expected result is achieved
	Erroneous - Solving a grid which has the start and destination cells connected to lines, but not a route from one to the other	When the solve button is pressed the route isn't displayed and the return button is shown	Expected result is achieved

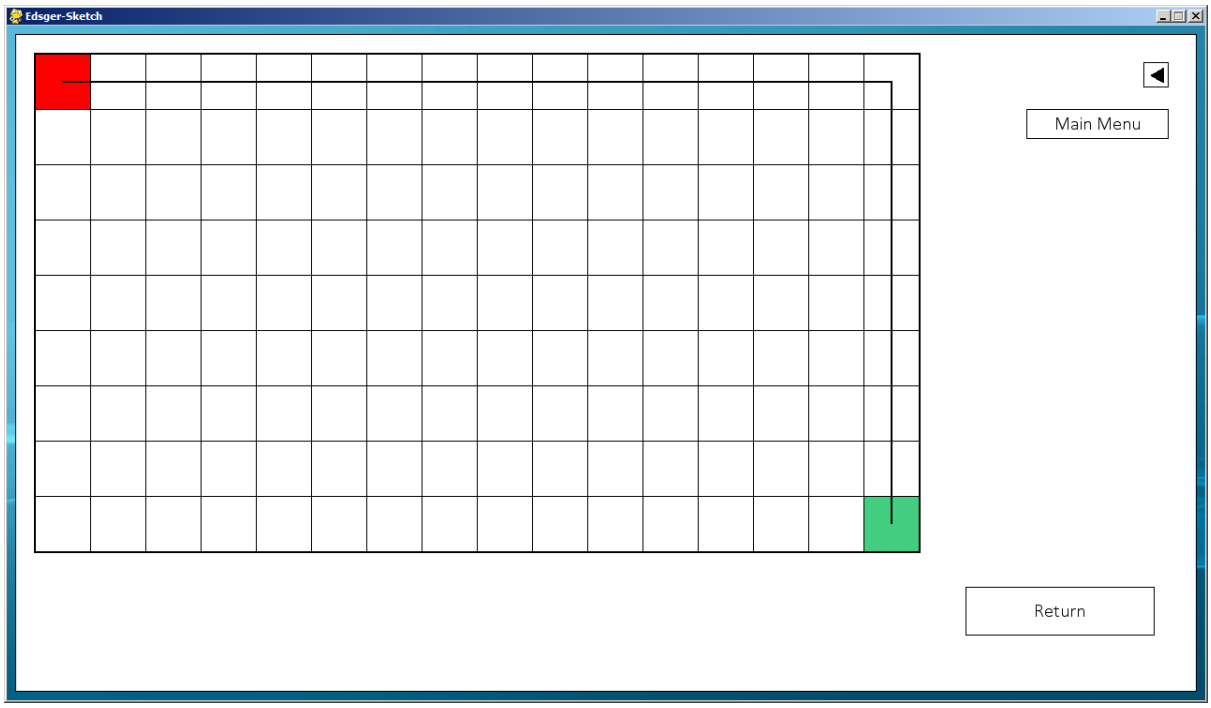
Here is the screen I will be working with for the grid function.



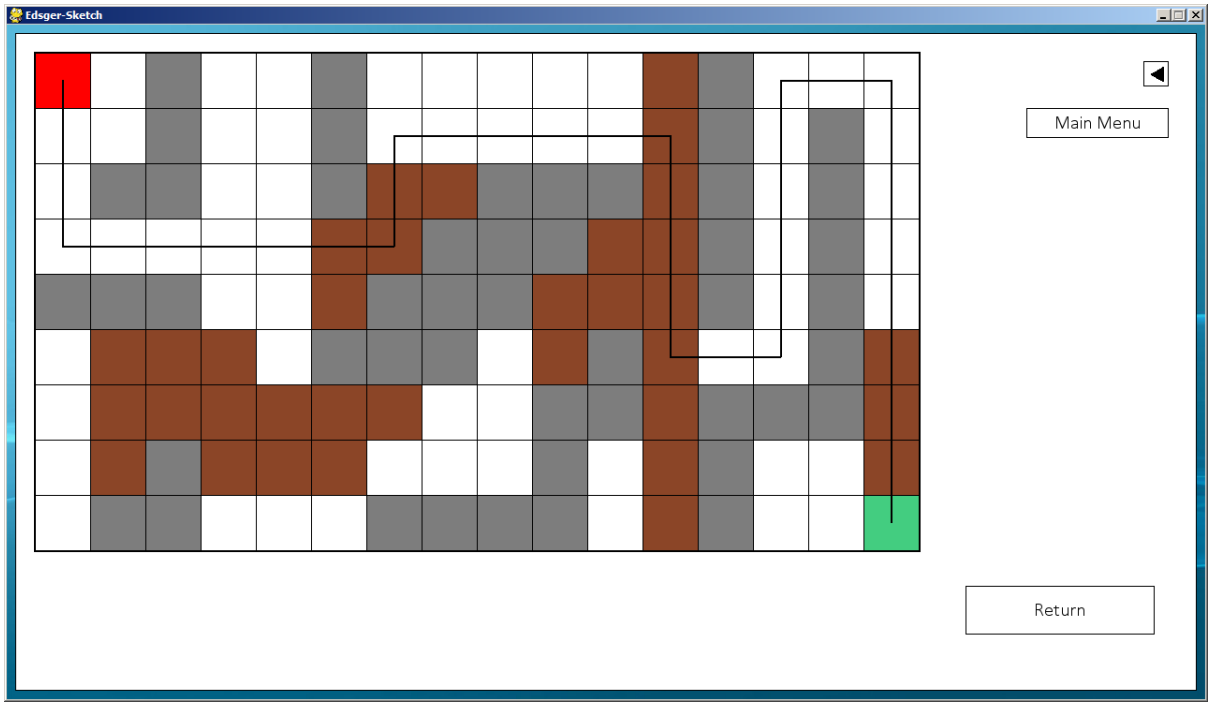
Here is the solve button being displayed since the start and destination cell are present.



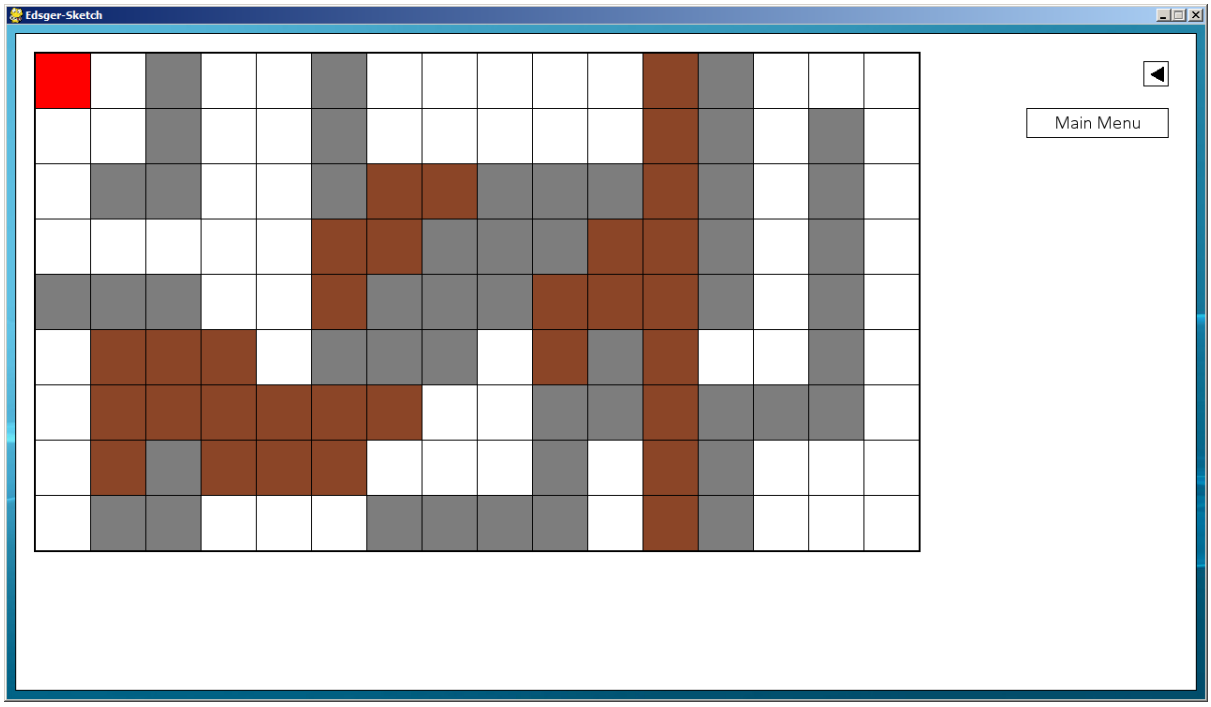
Here I solve the grid, showing just one of the shortest route, since there are many for a blank grid like this.



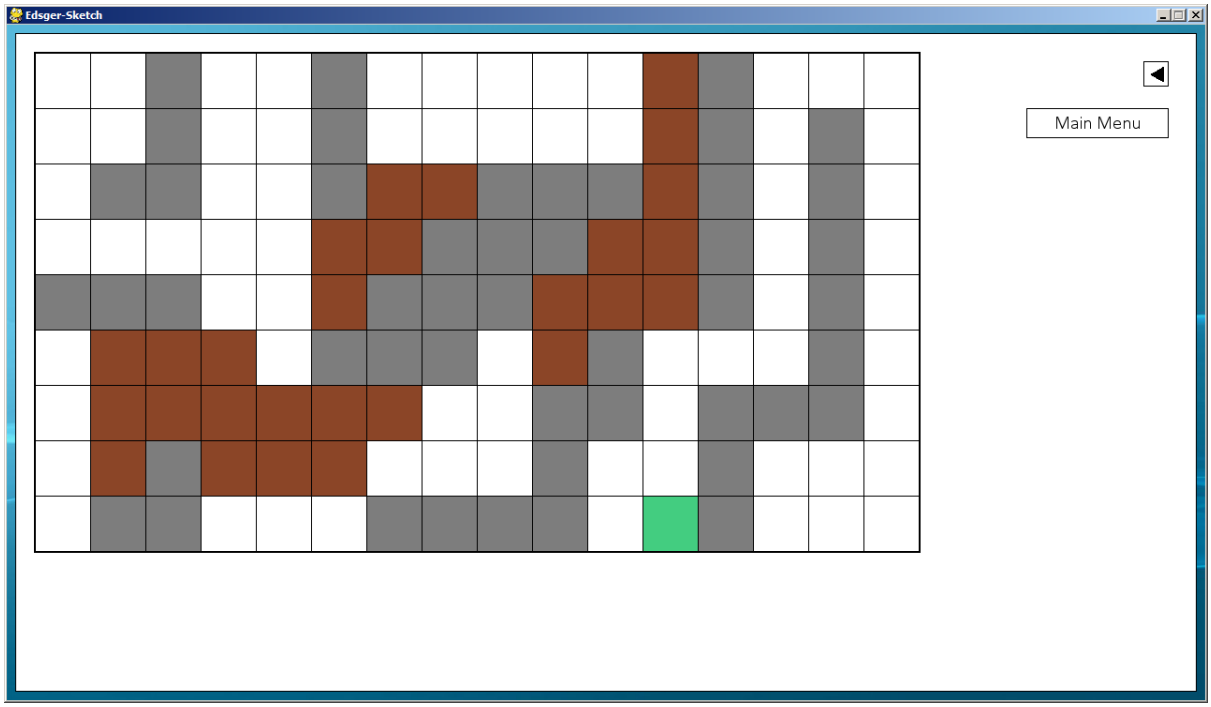
Here I test that it finds the shortest route in a more complicated grid, featuring slow terrain.



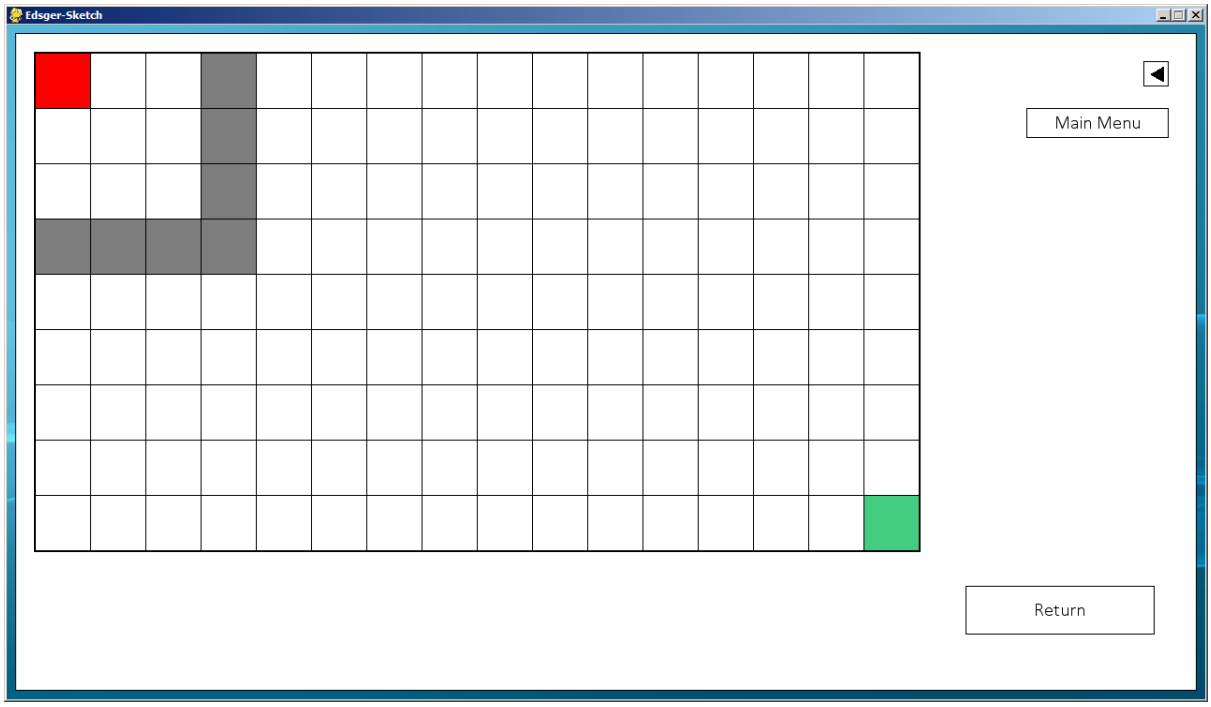
Here I test trying to solve the graph without a destination cell, the solve button doesn't appear since it is invalid.



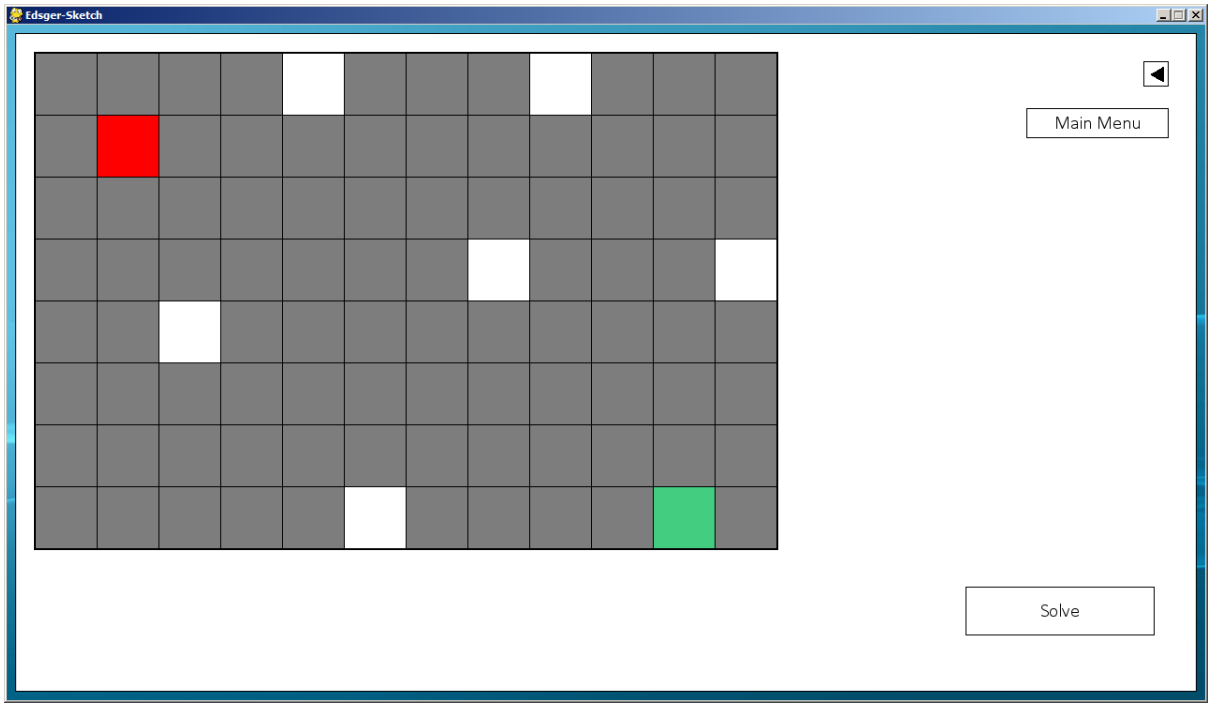
Here I test the same thing but without the start cell.



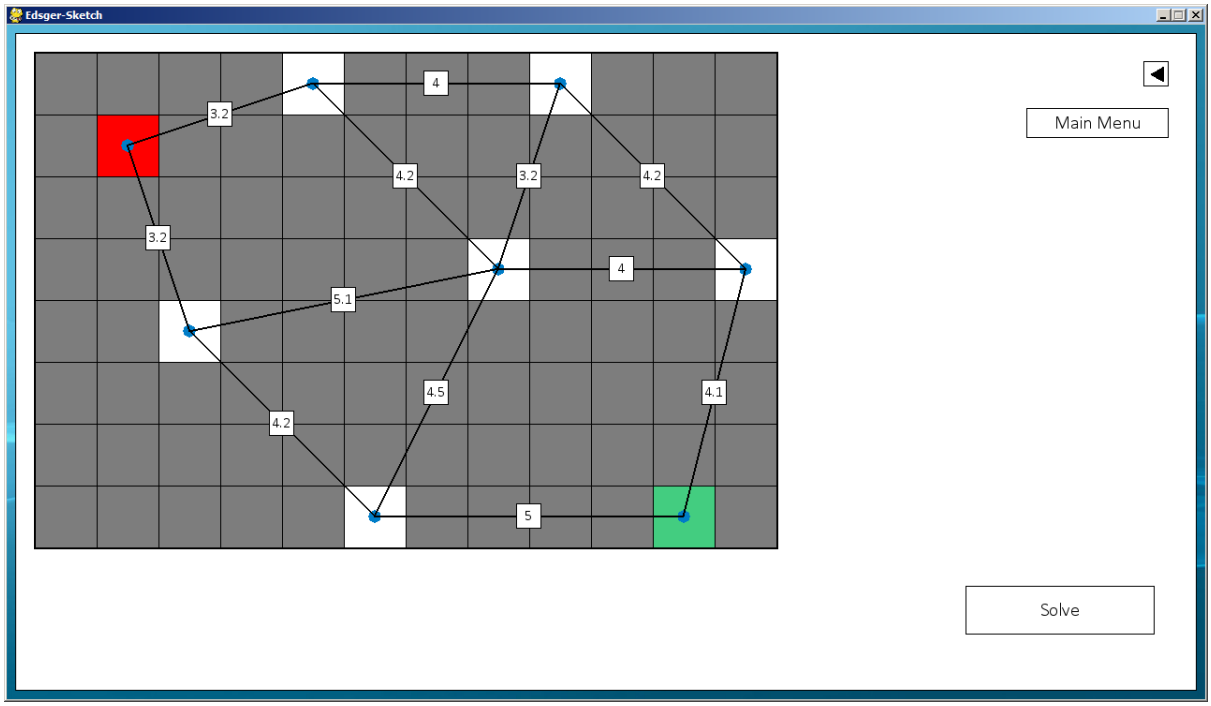
Here I test a grid that doesn't have a route to get from the start to the destination cell, no route is displayed.



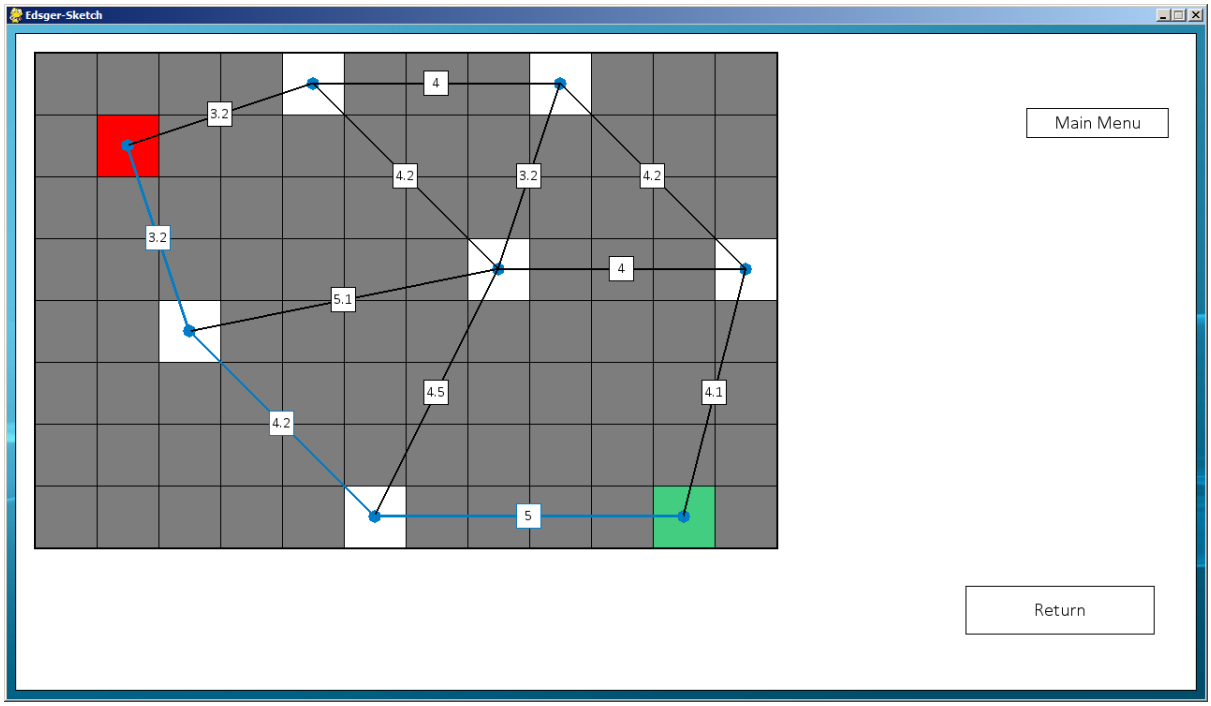
I now test the Vertices function; here is the screen I will be testing:



Here is the screen I will be testing for the Vertices function.

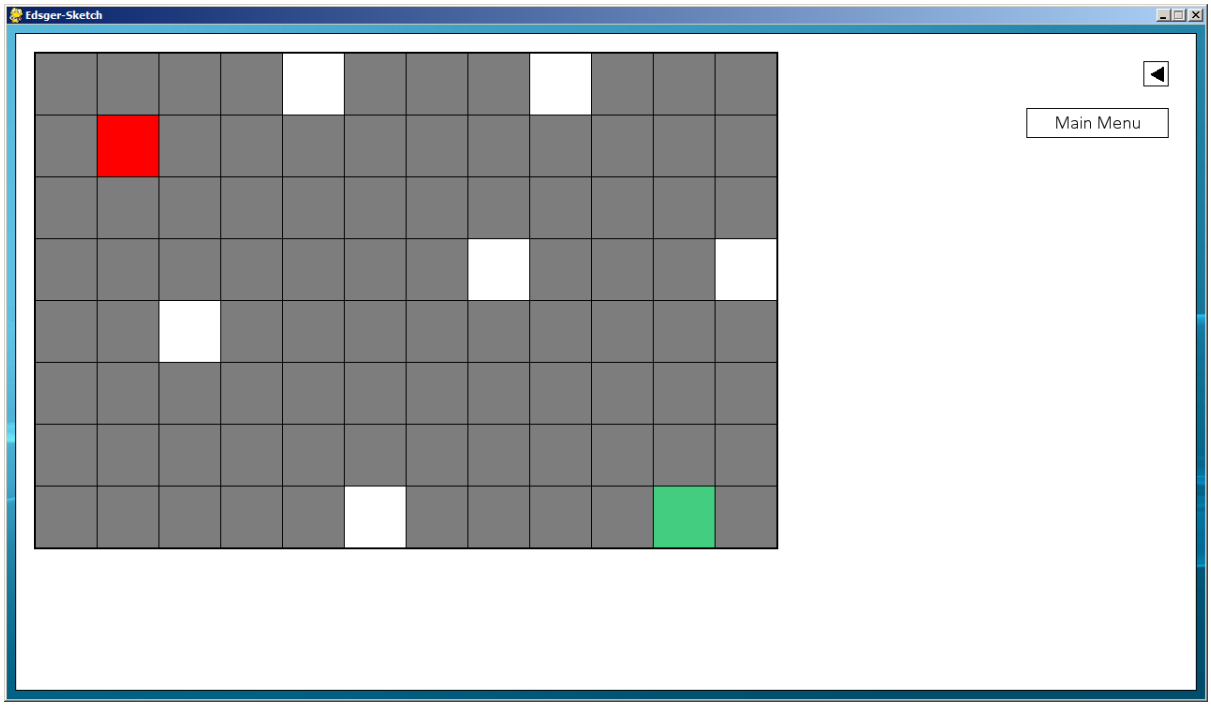


Here I test solving a valid graph.

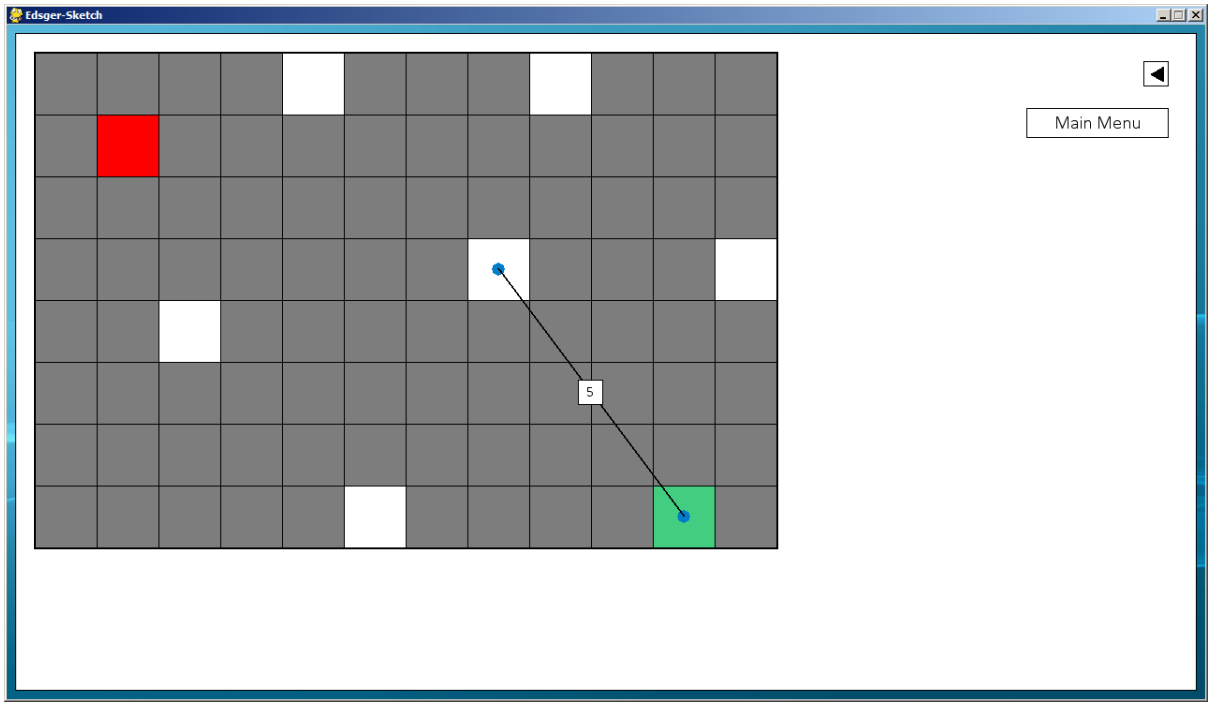


The shortest route is highlighted a light blue so the user can see it clearly.

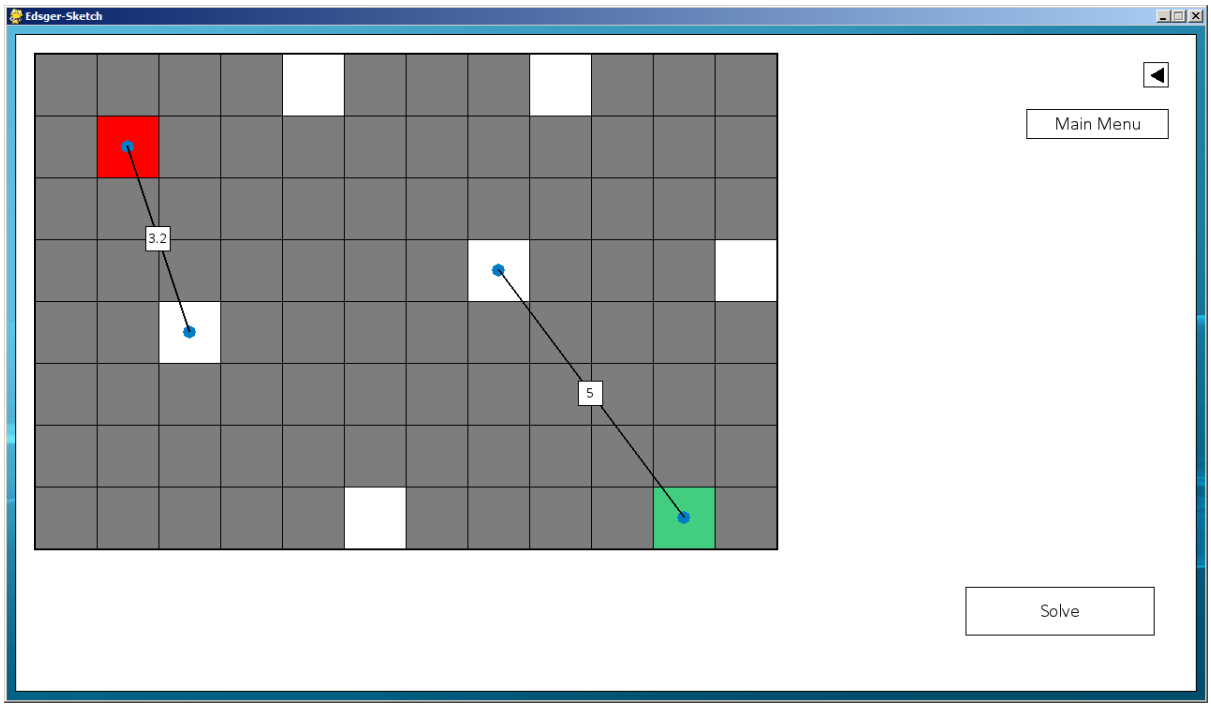
Here I test solving a grid without the start and destination cell connected to lines, the solve button isn't displayed.



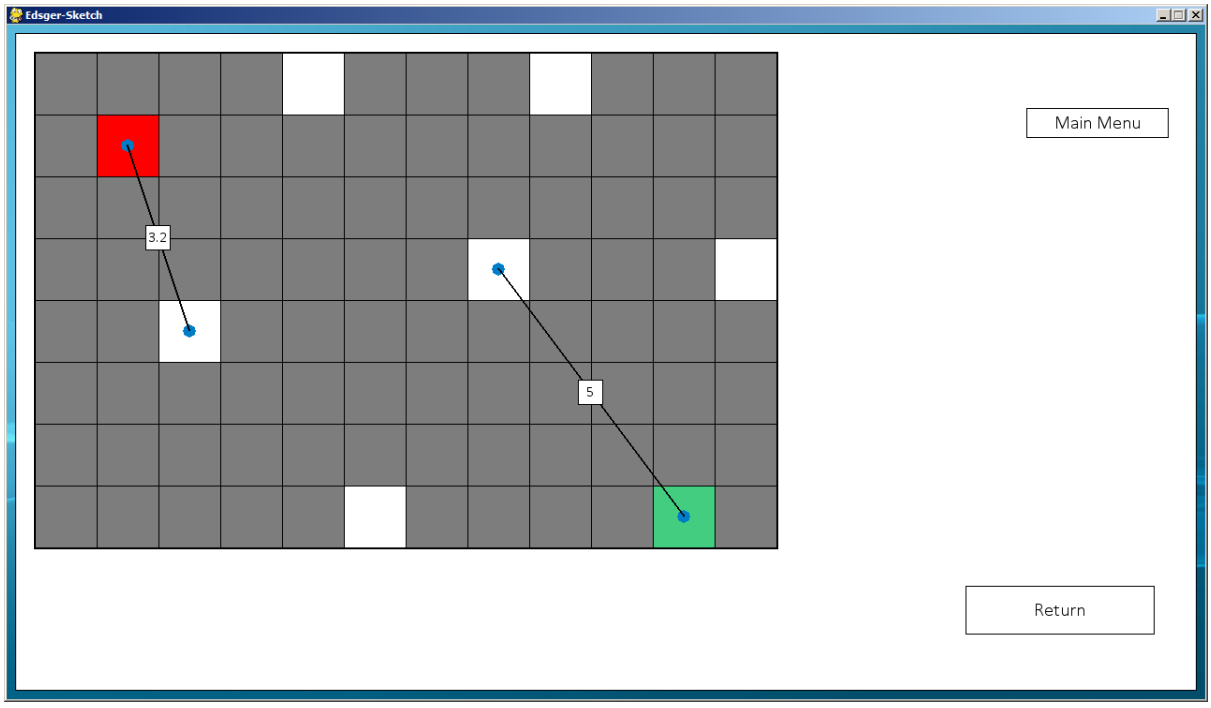
I then test with only the destination cell connected, it still doesn't show the solve button.



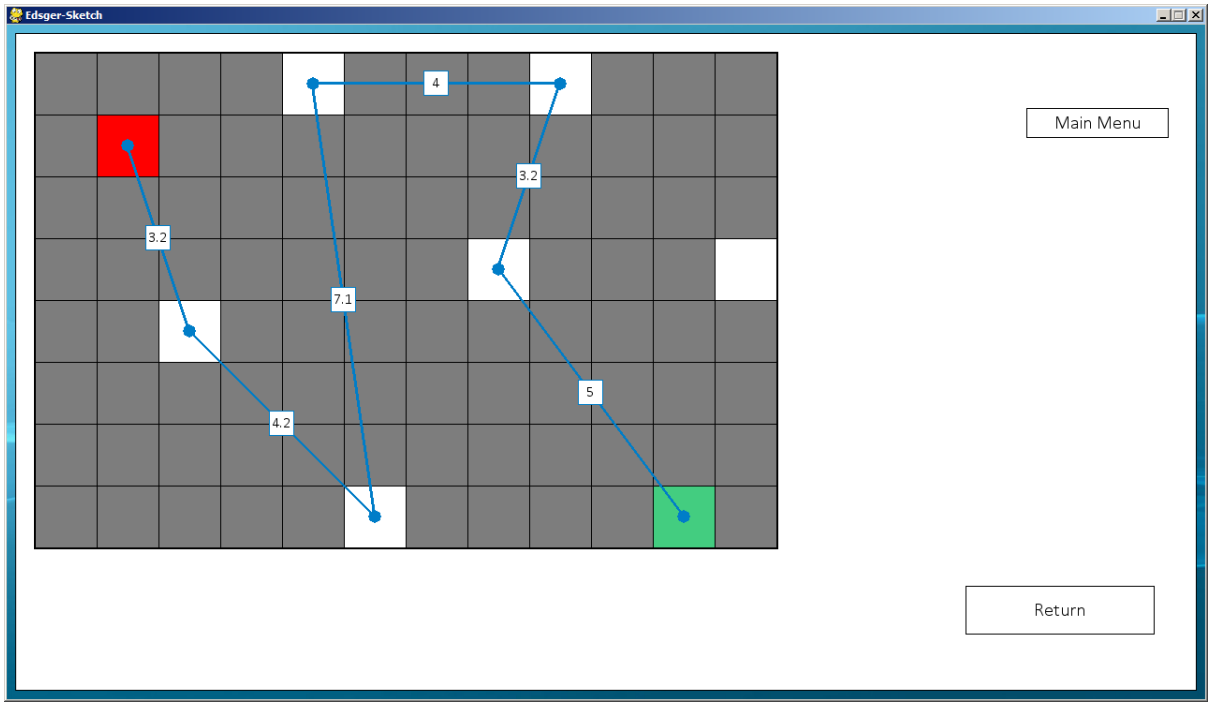
When the start and destination cell are connected to lines, the solve button is displayed.



However since there is no route between the two, it doesn't display the shortest route.



I then test that the graph still works even after being solved without a solvable route.



The program still solves the graph correctly.

System Maintenance

System Overview

My program is a learning resource to help both students to learn and teachers to teach Dijkstra's Algorithm. The first function displays a more practical application of the algorithm; it shows its potential uses so students can understand the sort of things the algorithm could be used for in real life situations. The second algorithm allows students and teachers to generate a weighted graph which is more alike to the sort of weighted graphs students might expect to find in their examination. Once the students have worked through the graph they can then solve the graph to see if they have also achieved the shortest route and compare their answers.

Fully Annotated Code

My annotated code is contained within the appendix, under the appropriate subheadings. The comments are shown by a hashtag (#); any characters which follow a hashtag on the same line will not be executed by the program and will only serve the purpose of being a comment. All comments are in green so they can be easily distinguished from the rest of the program.

List of all Functions/ Procedures and Variables

Functions are in bold and variables are not in bold.

Function/ Variable Name	Description
screen	Pygame window
bg_pic	Holds background image
background	Converted background image ready for use
black	RGB value for black
light_grey	RGB value for light grey
grey	RGB value for grey
white	RGB value for white
red	RGB value for red
green	RGB value for green
brown	RGB value for brown
light_blue	RGB value for light blue
options	Global dictionary holding
grid_references	
list_of_lines	
line_coordinates	
list_of_circles	

Sample Detailed Algorithm

Here I will cover, in detail, how the 'function2' function of the 'perform_dijkstras' file operates. I will cover each section of code step by step. This function assigns cumulative weights to each cell, which is the shortest route to that cell from the start position.

```
#This is the function that handles the Vertices function of our program
def function2(line_coordinates, start_cell, end_cell):
    #Creating place holder variables
    #Cells is a list of all the cells that are used in the lines - Not to
be mistaken with the line coordinates
    cells = []
    current_perm_value = 0
    finish = False
    valid = True
```

Here I define the function, with three variables in the brackets. These are sent in the main program when the function is called and provide all the necessary information to form the shortest route. The line_coordinates list contains the start and end coordinates of each line, along with the weight of the line. I then declare some variables. Cells will hold the detailed information about each cell, including the coordinates of it, the permanent value, the permanent cumulative weight and the working cumulative weight. The current_perm_value holds the current permanent value, this is a count that increments by one each time it loops through the main chunk of code to uniquely identify each cell. Finish controls when the loop will finish and valid is a validation check that changes if the route is invalid.

```
    #Multiply the weight of each line by 10 to avoid floating point
arithmetic errors
    for i in range(len(line_coordinates)):
        line_coordinates[i][4] = line_coordinates[i][4] * 10
```

Here I loop through every line in the line_coordinates list and multiply the weight of the line by 10. Since the weight of a line can only ever go to 1 decimal place, this means I multiply out the decimal so that I don't encounter any floating point arithmetic errors.

```
    #Creating a list of all the cells used in the program, with place
holder properties
    for i in line_coordinates:
        #Here we use the formula cells[column][row][0= x co-ordinate || 1=
y co-ordinate || 2= permanent value || 3= permanent cumulative weight || 4=
working cumulative weight] for retrieving data
        if [i[0], i[1], "", "", ""] not in cells:
            cells.append([i[0], i[1], "", "", ""])
        if [i[2], i[3], "", "", ""] not in cells:
            cells.append([i[2], i[3], "", "", ""])
```

Here I loop through all the lines and if the coordinates of one of the cells in the line, either the start or end coordinate of the line, isn't in the cells list, then I append it to the list. This means I create a list of every cell used by the lines, ready to assign properties to. I check both [i[0], i[1]] and [i[2], i[3]] since each line has two sets of coordinates.

```
    #Loop through all the cells
    for i in range(len(cells)):
        #Assign the start cell with a working cumulative weight of 0
        if cells[i][:2] == start_cell:
            cells[i][4] = 0
        #Retrieve the coordinates of the end cell
        elif cells[i][:2] == end_cell:
            end_cell_pos = i
```

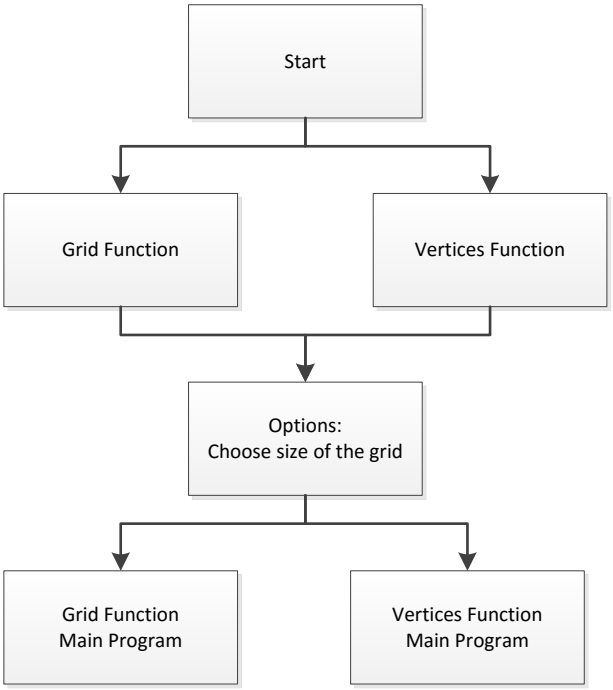
Here I loop through all the cells in my newly created cells list and check which one is the start and destination positions. For the start cell I assign it a permanent value of 0, since this is the start cell and is the first cell I will be working from. I also assign the position of the destination cell to a variable for later use.

```
    while finish != True:
```

```
        finish = ""
        current_perm_value += 1
```

Here I begin to run the main loop of this function. Finish is assigned to “”, which is changed to either False or True, depending on whether or not the loop has finished. Current_perm_value is a count which is assigned to a cell each time a permanent weight value is assigned, to show which order I assigned the values in.

Menu System



User Manual

Appendix

Main Program

Here is the fully annotated code for the main part of my program.

```
#Importing necessary modules
import pygame, sys, os, text_input, perform_dijkstras
from pygame.locals import *
from math import log10, floor
#Centring pygame window
os.environ['SDL_VIDEO_CENTERED'] = '1'
#Initialising pygame
pygame.init()
#Creating the display (resolution, flags, bits)
screen = pygame.display.set_mode((1280, 720), 0, 0)
#Setting title for the program
pygame.display.set_caption('Edsger-Sketch')
#Setting a background image
bg_pic = "background.jpg"
#Convert background image for use in program
background = pygame.image.load(bg_pic).convert()

#Declaring variables for RGB values of colours:
black = (0, 0, 0)
light_grey = (200, 200, 200)
grey = (125, 125, 125)
white = (255, 255, 255)
light_red = (255, 125, 125)
red = (255, 0, 0)
light_green = (100, 238, 161)
green = (67, 205, 128)
brown = (139, 69, 39)
light_blue = (0, 125, 200)

#Dictionary of global variables that holds the current settings of the
program
options = {'show_options':False, 'size_of_grid':[10, 7],
'selected_colour':white, 'size_of_square':66, 'show_solve_button':False,
'solve':False, 'show_options_arrow':True, 'function':2,
'start_of_line':True, 'lines':False}
#Since this options dictionary is a global variable, it means any function
can access it, so it serves as a way for the program to communicate with
itself. When a value is changed, such as the size of the grid, the part of
the program which changes that value will change the corresponding value in
the dictionary. Then when running a function that displays and updates
pygame window, that function will retrieve the necessary information from
this dictionary. So for example when running the function which displays
the grid, to find out the how many cells it needs to display, it will
retrieve that information from this dictionary.

#Declaring empty global lists for use in the program:
grid_references = []
list_of_lines = []
line_coordinates = []
list_of_circles = []

#The first screen of the program
def main_menu():
```

```
    #Standard use of the box class to create text boxes or buttons on
screen with text in them:
    #blit the background image onto the screen of the program
    screen.blit(background,(0,0))
    #Assigning variable to instance of the class 'box'. The box has default
values for each variable, so if a variable isn't declared it will use the
standard ones
    start_button = box()

    #Setting the size of the box (x, y)
    start_button.size = (300, 125)
    #Setting the location for the top left of the box (x, y)
    start_button.location = (490, 250)
    #Declaring the size of the font for the text
    start_button.font_size = 22
    #The size, in pixels, of the black outline for the box.
    start_button.size_outline = 2
    #The text to be featured in the box, which is in this case the start
button
    start_button.text_input = "Start"
    #Necessary modules called to create the box. Draw box displays the box
with it's outline on screen, prep creates the font and initialises it. This
is separate module in case I need to create a box without text. Centralise
text centralises the text in both the x and y directions. Display text
displays the text on top of the box, this is a separate module since
sometimes I won't want to centralise the text.
    start_button.draw_box(), start_button.prep(),
start_button.centralise_text(), start_button.display_text()
    #Getting the coordinates in the format (Start x coordinate (left), end
x coordinate (right), start y coordinate (top), end y coordinate (bottom).
These are used to detect which button has been clicked, if any, when the
mouse button is pressed
    sb_co = start_button.get_coords()

    #Infinite loop - needs to be infinite to constantly update the pygame
window
    while True:
        #Get the x and y coordinates of the mouse relative to the pygame
window
        x,y = pygame.mouse.get_pos()
        #Loops through the events that have backlogged since the last time
it ran through this loop - events are actions such as the mouse button
being pressed or a key on the keyboard being pressed etc.
        for event in pygame.event.get():

            #If the event type is QUIT, meaning the X button has been
pressed at the top right of the window, or the escape button is pressed:
            if event.type == QUIT or (event.type == KEYDOWN and event.key
== K_ESCAPE):
                #Quit pygame
                pygame.quit()
                #Also exit the system to make sure there isn't an empty
shell of a pygame window left running
                sys.exit()

            #If the event is the mouse button being released - Pushing back
up after being pressed down
            if event.type == MOUSEBUTTONUP:
                #If the coordinates of the mouse when the mouse button was
released are within the coordinates sb_co (The start button)
```

```
                if (x >= sb_co[0] and x <= sb_co[1]) and (y >= sb_co[2] and
y <= sb_co[3]):
                    #Then perform the choose_function function - Displays
the next screen
                    choose_function()

        #Update the pygame window with any changes. Needs to be called
otherwise the screen will never change.
        pygame.display.update()

#Choose which function of the program the user wants to perform.
def choose_function():
    #Display background
    screen.blit(background,(0,0))

    #Defining variables for instance of box class - same routine as when
creating the start box:
    text_box1 = box()
    #This runs the default_text_box module, which changes some of the
default values for variables which are more suited to a text box.
    text_box1.default_text_box()
    text_box1.text_location = (0, 465)
    text_box1.text_input = "Hover over a button for a more detailed
description."
    #Here I only centralise the text in the x direction since I want the
text to sit at the top of the box since it's a text box not a button
    text_box1.draw_box(), text_box1.prep(), text_box1.centralise_x()
,text_box1.display_text()

    #Defining variables for a button instance of box class:
    button1 = box()
    #This runs the default_button module, which changes some of the default
values for variables which are more suited to a button.
    button1.default_button()

    button1.location = (400, 200)
    button1.text_input = "Grid"
    button1.draw_box(), button1.prep(), button1.centralise_text(),
button1.display_text()
    b1co = button1.get_coords()

    #Defining variables for another button:
    button2 = box()
    button2.default_button()
    button2.location = (730, 200)
    button2.text_input = "Vertices"
    button2.draw_box(), button2.prep(), button2.centralise_text(),
button2.display_text()
    b2co = button2.get_coords()

    #Infinite loop
    while True:
        #Mouse coordinates
        x, y = pygame.mouse.get_pos()
        #Event handling loop
        for event in pygame.event.get():

            #Quitting the program
            if event.type == QUIT or (event.type == KEYDOWN and event.key
== K_ESCAPE):
                pygame.quit()
```

```
                sys.exit()

            #Check if button 1 has been pressed
            if (x >= b1co[0] and x <= b1co[1]) and (y >= b1co[2] and y <=
b1co[3]):
                #When hovering over a button, without pressing it, change
the text of the text box to this
                text_box1.text_input = "All cells are available. Choose the
type of terrain for each cell and it will find the shortest route."

                if event.type == MOUSEBUTTONUP:
                    #Set function part of options to 1, so after the
options screen it knows which function to perform
                    options['function'] = 1
                    #Perform options_screen function - Display the options
screen
                    options_screen()

            #Check if button 2 has been pressed
            elif (x >= b2co[0] and x <= b2co[1]) and (y >= b2co[2] and y <=
b2co[3]):
                #When hovering over a button, without pressing it, change
the text of the text box to this
                text_box1.text_input = "Only selected cells are available.
More standard Dijkstra's Algorithm operations with weighted vertices."

                if event.type == MOUSEBUTTONUP:
                    #Set function part of options to 1, so after the
options screen it knows which function to perform
                    options['function'] = 2
                    #Perform options_screen function - Display the options
screen
                    options_screen()

            else:
                #If no buttons are currently being hovered over, change the
text of the text box to this
                text_box1.text_input = "Hover over a button for a more
detailed description."

            #Re draw the text box to erase the old text and display the
updated text.
            text_box1.draw_box(), text_box1.prep(),
text_box1.centralise_x() ,text_box1.display_text()

        pygame.display.update()

#Options screen before accessing the main program.
def display_options():
    #Declaring empty coordinates list to append the coordinates of each
button to later
    coordinates = []
    #Blit background
    screen.blit(background,(0,0))

    #Creating all the text boxes and buttons necessary:
    options_area = box()
    options_area.location = (440, 200)
    options_area.size = (400, 200)
    options_area.font_size = 18
    options_area.text_location = (450, 210)
```

```
    options_area.text_input = "Options:"
    options_area.draw_box(), options_area.prep(),
options_area.display_text()

    #Here I am assigning a font to a variable for use later on
    font = pygame.font.SysFont("calibri", 18)
    #I am using the font to display the text "Size of Grid:" on screen
    screen.blit(font.render("Size of Grid:", 1, black), (520, 275))
    size_of_grid = box()
    size_of_grid.default_options_buttons()
    size_of_grid.location = (640, 270)
    size_of_grid.size = (35, 25)
    #The text for this box is the current x value for the size of the grid
- This is so it automatically updates when the user enters a value
    size_of_grid.text_input = str(options['size_of_grid'][0])

    size_of_grid.font_size = 16
    size_of_grid.draw_box(), size_of_grid.prep(),
size_of_grid.centralise_text(), size_of_grid.display_text()
    #Appending the coordinates of the this button to the list
    coordinates.append(size_of_grid.get_coords())

    screen.blit(font.render("x", 1, black), (690, 275))
    size_of_grid2 = box()
    size_of_grid2.default_options_buttons()
    size_of_grid2.location = (715, 270)
    size_of_grid2.size = (35, 25)
    size_of_grid2.text_input = str(options['size_of_grid'][1])
    size_of_grid2.font_size = 16
    size_of_grid2.draw_box(), size_of_grid2.prep(),
size_of_grid2.centralise_text(), size_of_grid2.display_text()
    coordinates.append(size_of_grid2.get_coords())

    #Assigning text options to a variable
    font = pygame.font.SysFont("calibri", 16)
    #Assigning text to a variable
    max_size1 = font.render("Max Size of Grid:", 1, black)
    #Displaying text on screen
    screen.blit(max_size1, (520, 310))

    #If running the first function
    if options['function'] == 1:
        #The max x value is 40
        max_size2 = font.render("40", 1, black)
    #If running the second function
    elif options['function'] == 2:
        #The max x value is 14
        max_size2 = font.render("14", 1, black)
    #Displaying the max x value
    screen.blit(max_size2, (650, 310))

    #Assigning text and options together in a variable
    max_size3 = pygame.font.SysFont("calibri", 20).render("x", 1, black)
    #Displaying text max_size3 on screen
    screen.blit(max_size3, (690, 307))

    #If running the first function
    if options['function'] == 1:
        #The max y value is 30
        max_size4 = font.render("30", 1, black)
    #If running the second function
```

```
    elif options['function'] == 2:
        #The max y value is 10
        max_size4 = font.render("10", 1, black)
    #Display the max y value
    screen.blit(max_size4, (725, 310))

    finished = box()
    finished.location = (750, 365)
    finished.size = (80, 25)
    finished.font_size = 18
    finished.text_input = "Finished"
    finished.draw_box(), finished.prep(), finished.centralise_text(),
finished.display_text()
    coordinates.append(finished.get_coords())
    #Return the coordinate list, containing the coordinates of any buttons
necessary for use
    return coordinates

def options_screen():
    #Display all the text boxes and buttons in the display_options
function, and assign the coordinates list to a variable
    coordinates = display_options()
    #First size of grid box is the first set of coordinates in the list
    sog_co = coordinates[0]
    #Second size of grid box is the second set of coordinates
    sog2_co = coordinates[1]
    #Third set of coordinates is the finish button
    fin_co = coordinates[2]

    #Infinite loop
    while True:
        x, y = pygame.mouse.get_pos()
        #Event handling loop
        for event in pygame.event.get():
            if event.type == QUIT or (event.type == KEYDOWN and event.key
== K_ESCAPE):
                #Exiting program
                pygame.quit()
                sys.exit()

            if event.type == MOUSEBUTTONUP:
                #If the mouse is inside the first size of grid box during
the MOUSEBUTTONUP event:
                if (x >= sog_co[0] and x <= sog_co[1]) and (y >= sog_co[2]
and y <= sog_co[3]):
                    #Coordinates of the first size of grid box - the x
value (startx, starty, endx, endy)
                    coords = (640, 270, 675, 295)
                    #Size of the box (x, y)
                    size = (35, 25)

                    #Call the text_input_display function from the
text_input imported file to allow the user to type on screen in the size of
grid text input
                    size_of_grid_input =
text_input.text_input_display(screen, coords, size)
                    #Assigning the full text input to a variable
                    width_value = size_of_grid_input.completed_text

                    #If the length of the text doesn't equal 0 (they have
actually input something)
```

```
                    if len(width_value) != 0:
                        #Turn the input to an integer - If they didn't
input anything it would return an error so this comes after checking the
length of the input - Validation to check if the input is a number is
inside the text_input file
                        width_value = int(width_value)

                        #If the value is less than 0
                        if width_value <= 0:
                            #Change it to the lowest accepted value, which
is 1
                            width_value = 1

                        #If function 1 is running, the max x value is 40,
so if it is greater than that
                        elif options['function'] == 1 and width_value > 40:
                            #Change the value to the highest accepted
value, which is 40
                            width_value = 40

                        #If function 2 is running, the max x value is 14,
so if it is greater than that
                        elif options['function'] == 2 and width_value > 14:
                            #Change the value to the highest accepted
value, which is 14
                            width_value = 14

                        #Once the validation is complete, update the size
of grid value in the options dictionary to reflect the change
                        options['size_of_grid'][0] = width_value

                #This is the same as above, but this time for the second
size of grid box, which holds the y value
                elif (x >= sog2_co[0] and x <= sog2_co[1]) and (y >=
sog2_co[2] and y <= sog2_co[3]):
                    coords = (715, 270, 750, 295)
                    size = (35, 25)
                    size_of_grid2_input =
text_input.text_input_display(screen, coords, size)
                    height_value = size_of_grid2_input.completed_text
                    if len(height_value) != 0:
                        height_value = int(height_value)
                        if height_value <= 0:
                            height_value = 1
                        elif options['function'] == 1 and height_value >
30:
                            height_value = 30
                        elif options['function'] == 2 and height_value >
10:
                            height_value = 10
                        options['size_of_grid'][1] = height_value

                #If the mouse is over the finish button during the
MOUSEBUTTONUP event:
                elif (x >= fin_co[0] and x <= fin_co[1]) and (y >=
fin_co[2] and y <= fin_co[3]):
                    #Run the set_square_value function, to find the size of
the square, commented further down:
                    set_square_value()
```

```
                    #Detects which function is currently selected from the
options dictionary and runs the appropriate function (the next page):
                    if options['function'] == 1:
                        function1()
                    elif options['function'] == 2:
                        function2()

        #This runs through the display options function again to update any
value that have changed, for example the size of the grid
        display_options()
        #Update the pygame display
        pygame.display.update()

def set_square_value():
    #Since the maximum number of pixels the width of the grid can take up
is 950, I divide this to get what would be the number of pixels for each
square, rounded down. For example if I had 10 cells along the x axis, each
cell would be 95 pixels wide (however this would be changed to 66 since
that is the maximum pixel number for a cell)
    temp_square_value = 950/options['size_of_grid'][0]
    #The same process but now for the number of rows (number of cells going
down/ on the y axis)
    temp_square_value2 = 660/options['size_of_grid'][1]

    #This finds out which of the two pixel sizes for the cells is smaller:
    if temp_square_value <= temp_square_value2:
        square_value = temp_square_value
    elif temp_square_value2 <= temp_square_value:
        square_value = temp_square_value2

    #The maximum number of pixels for a cell is 66, so if it is greater
than this it changes it back to 66
    if square_value > 66:
        square_value = 66

    #Sets the size of the square to the now validated value
    options['size_of_square'] = square_value

#Function 1 - Grid function
def function1():
    #Setting the function to use the global variable "grid_references"
instead of creating a new local one
    global grid_references
    #Blit background
    screen.blit(background,(0,0))

    #Assigns an instance of the function1_screen class to window, this is
because certain coordinates are needed before the first update
    window = function1_screen()
    #Running the module draw_initial_screen
    window.draw_initial_screen()

    #It fetches these coordinates first since they can be pressed before
the screen is updated for the first time
    coordinate_list = window.coordinate_list
    arrow_co = window.arrow_box_coordinates
    home_co = window.home_button_coordinates

    #Infinite loop
    while True:
        x, y = pygame.mouse.get_pos()
```

```
        #Event handling loop
        for event in pygame.event.get():
            if event.type == QUIT or (event.type == KEYDOWN and event.key
== K_ESCAPE):
                #Quit program
                pygame.quit()
                sys.exit()

            if event.type == MOUSEBUTTONUP:
                #This checks if the mouse is within the arrow that can be
seen in the top right corner of the program
                if (x >= arrow_co[0] and x <= arrow_co[1]) and (y >=
arrow_co[2] and y <= arrow_co[3]) and options['show_options_arrow'] ==
True:
                    #If the options screen isn't showing already then
display it:
                    if options['show_options'] == False:
                        window = function1_screen()
                        window.draw_options_screen()
                        #Retrieving necessary coordinates
                        coordinate_list = window.coordinate_list
                        #Changing options dictionary to show that the
options screen is currently displaying
                        options['show_options'] = True

                    #If the options screen is showing then hide it:
                    elif options['show_options'] == True:
                        window = function1_screen()
                        window.draw_no_options_screen()
                        coordinate_list = window.coordinate_list
                        options['show_options'] = False

                #If the options are showing, then the cells can be edited:
                if options['show_options'] == True:
                    #Assigning each set of coordinates to a variable
                    grid_co = coordinate_list[0]    #grid co-ordinates
                    sog_co = coordinate_list[1]     #size of grid co-
ordinates
                    sog2_co = coordinate_list[2]    #size of grid 2nd box
co-ordinates
                    sp_co = coordinate_list[3]      #starting position co-
ordinates
                    pt_co = coordinate_list[4]      #passable terrain co-
ordinates
                    it_co = coordinate_list[5]      #impassable terrain co-
ordinates
                    dp_co = coordinate_list[6]      #destination position
co-ordinates
                    st_co = coordinate_list[7]      #slow terrain co-
ordinates

                    #If the size of grid box has been clicked, retrieve and
display the input text and update option dictionary to reflect these
changes
                    if (x >= sog_co[0] and x <= sog_co[1]) and (y >=
sog_co[2] and y <= sog_co[3]):
                        coords = (1150, 95, 1180, 115)
                        size = (30, 20)
                        size_of_grid_input =
text_input.text_input_display(screen, coords, size)
                        width_value = size_of_grid_input.completed_text
```

```
                        if len(width_value) != 0:
                            width_value = int(width_value)
                            if width_value <= 0:
                                width_value = 1
                            elif width_value > 40:
                                width_value = 40
                            options['size_of_grid'][0] = width_value
                        set_square_value()

                    #If the size of grid box has been clicked, retrieve and
display the input text and update option dictionary to reflect these
changes
                    elif (x >= sog2_co[0] and x <= sog2_co[1]) and (y >=
sog2_co[2] and y <= sog2_co[3]):
                        coords = (1210, 95, 1240, 115)
                        size = (30, 20)
                        size_of_grid2_input =
text_input.text_input_display(screen, coords, size)
                        height_value = size_of_grid2_input.completed_text
                        if len(height_value) != 0:
                            height_value = int(height_value)
                            if height_value <= 0:
                                height_value = 1
                            elif height_value > 30:
                                height_value = 30
                            options['size_of_grid'][1] = height_value
                        set_square_value()

                    #If one of the options has been clicked, it updates the
currently selected colour in the options dictionary
                    elif (x >= sp_co[0] and x <= sp_co[1]) and (y >=
sp_co[2] and y <= sp_co[3]):
                        options['selected_colour'] = red
                    elif (x >= pt_co[0] and x <= pt_co[1]) and (y >=
pt_co[2] and y <= pt_co[3]):
                        options['selected_colour'] = white
                    elif (x >= st_co[0] and x <= st_co[1]) and (y >=
st_co[2] and y <= st_co[3]):
                        options['selected_colour'] = brown
                    elif (x >= it_co[0] and x <= it_co[1]) and (y >=
it_co[2] and y <= it_co[3]):
                        options['selected_colour'] = grey
                    elif (x >= dp_co[0] and x <= dp_co[1]) and (y >=
dp_co[2] and y <= dp_co[3]):
                        options['selected_colour'] = green

                    #When the mouse is clicked within the grid it takes the
selected colour/ property and applies it to the cell that was clicked
                    elif (x >= grid_co[0] and x <= grid_co[1]) and (y >=
grid_co[2] and y <= grid_co[3]):
                        #Retrieving the number of pixels in each cell from
the options
                        size_of_square = options['size_of_square']
                        #This takes the x and y value of the mouse and
subtracts 30 from it, since the start of the grid, the top left of the
grid, is 30 pixels away from the edge of the pygame in both axes. In other
words the grid starts at (30, 30) so it find the mouse position relative to
the start of the grid
                        temp_x, temp_y = x - 30, y - 30
```

```
                        #Takes the pixel coordinates of the mouse and
divides them by the pixel size of each cell to find which cell it's in
                        row = temp_x / size_of_square
                        column = temp_y / size_of_square

                        #This checks if the mouse is on the line in between
the cells or actually on a cell, if it's on the line it doesn't process it
                        if (temp_x % size_of_square) != 0 and (temp_y %
size_of_square) != 0:

                            #If the selected colour is green or red
                            if options['selected_colour'] == green or
options['selected_colour'] == red:
                                #Loop through the grid
                                for i in range(options['size_of_grid'][0]):
                                    for j in
range(options['size_of_grid'][1]):
                                        #If the selected colour is green
and it finds another green cell within the grid
                                        if grid_references[i][j][2] ==
green and options['selected_colour'] == green:
                                            #Change the old green cell to
white, since there can only be one start cell
                                            grid_references[i][j][2] =
white
                                        #The same but with the destination
cell/ red colour
                                        elif grid_references[i][j][2] ==
red and options['selected_colour'] == red:
                                            grid_references[i][j][2] =
white

                            #Applies the colour to the appropriate cell
                            grid_references[row][column][2] =
options['selected_colour']
                            #It updates the grid_references since when the
grid updates it will retrieve the colour of the cell from the
grid_references

                    #Updates the screen, using the module that includes the
options screen
                    window = function1_screen()
                    window.draw_options_screen()
                    coordinate_list = window.coordinate_list

                #If the options screen isn't showing: (When the options
screen isn't up the solve and home buttons are available)
                elif options['show_options'] == False:
                    #If the home button is pressed
                    if (x >= home_co[0] and x <= home_co[1]) and (y >=
home_co[2] and y <= home_co[3]):
                        #Erase grid references to stop cells carrying over
to a new grid
                        grid_references = []
                        options['solve'] = False
                        options['show_options_arrow'] = True
                        #Return to the coose_function screen
                        choose_function()

                    #If the solve button is showing (basic validation
checking if the start and destination coordinates are on the grid)
```

```
                    elif options['show_solve_button'] == True:
                        #Solve button co-ordinates - the coordinates list
is different for the screen without options
                        sb_co = coordinate_list[1]

                        #If the solve button is pressed
                        if (x >= sb_co[0] and x <= sb_co[1]) and (y >=
sb_co[2] and y <= sb_co[3]):

                            #If the graph isn't currently solved
                            if options['solve'] == False:
                                #Changing options to show the graph is
solved
                                options['solve'] = True
                                #Removes the options arrow while the graph
is solved
                                options['show_options_arrow'] = False

                            #If the graph isn't solved the text is now
"Return" and it removes the solution from the screen
                            elif options['solve'] == True:
                                options['solve'] = False
                                options['show_options_arrow'] = True

                        #Updates screen
                        window.draw_no_options_screen()
                        coordinate_list = window.coordinate_list

        pygame.display.update()

#Function 2
def function2():
    #These variables are globalised, since they need to be set to empty
lists when the home button is pressed and if it isn't globalised it tries
to create new local variables
    global grid_references, list_of_lines, list_of_circles,
line_coordinates, final_route

    #Resetting the Solve button to False, in case they came from a solved
function
    options['show_solve_button'] = False

    #Blit background
    screen.blit(background,(0,0))
    #Assigning variable
    initial_screen = function1_screen()
    #Displaying the screen
    initial_screen.draw_initial_screen()

    #Assigning coordinates to variables
    coordinate_list = initial_screen.coordinate_list
    arrow_co = initial_screen.arrow_box_coordinates
    home_co = initial_screen.home_button_coordinates

    #These are the start and end cells when retrieved by a line. If a line
is drawn and it starts or ends at either the start of destination position
these variables will be filled in. It provides a simple form a validation.
    start_cell, end_cell = "", ""

    #This is a variable that holds the coordinates of the start cell of a
line if a line is being drawn
```

```
    start_grid = ""

    #Infinite loop
    while True:
        x, y = pygame.mouse.get_pos()
        #Event handling loop
        for event in pygame.event.get():
            if event.type == QUIT or (event.type == KEYDOWN and event.key
== K_ESCAPE):
                #Quit program
                pygame.quit()
                sys.exit()

            if event.type == MOUSEBUTTONUP:
                #Displaying/ hiding options
                if (x >= arrow_co[0] and x <= arrow_co[1]) and (y >=
arrow_co[2] and y <= arrow_co[3]) and options['show_options_arrow'] ==
True:
                    if options['show_options'] == False:
                        options_screen = function1_screen()
                        options_screen.draw_options_screen()
                        coordinate_list = options_screen.coordinate_list
                        options['show_options'] = True

                    elif options['show_options'] == True:
                        options['start_of_line'] = True
                        if start_grid != "":

grid_references[start_grid[0]][start_grid[1]][2] = start_colour
                        no_options_screen = function1_screen()
                        no_options_screen.draw_no_options_screen()
                        coordinate_list = no_options_screen.coordinate_list
                        options['show_options'] = False

                #If options are displayed, enable ability to edit grid
                if options['show_options'] == True:
                    #Assigning coordinates of each button to a variable
                    grid_co = coordinate_list[0]    #grid co-ordinates
                    sog_co = coordinate_list[1]     #size of grid co-
ordinates
                    sog2_co = coordinate_list[2]    #size of grid 2nd box
co-ordinates
                    sp_co = coordinate_list[3]      #starting position co-
ordinates
                    pt_co = coordinate_list[4]      #passable terrain co-
ordinates
                    it_co = coordinate_list[5]      #impassable terrain co-
ordinates
                    dp_co = coordinate_list[6]      #destination position
co-ordinates
                    dl_co = coordinate_list[7]      #draw lines button co-
ordinates
                    #If there are lines on screen then assign the clear
lines button coordinates, if there are no lines this button isn't displayed
                    if len(list_of_lines) != 0:
                        cl_co = coordinate_list[8]  #clear lines button co-
ordinates

                    #Whilst there aren't any lines on the screen - Since
options are disabled when a line has been drawn
                    if len(list_of_lines) == 0:
```

```
                        #Retrieve and display number of columns input
                        if (x >= sog_co[0] and x <= sog_co[1]) and (y >=
sog_co[2] and y <= sog_co[3]):
                            coords = (1150, 95, 1180, 115)
                            size = (30, 20)
                            size_of_grid_input =
text_input.text_input_display(screen, coords, size)
                            width_value = size_of_grid_input.completed_text
                            if len(width_value) != 0:
                                width_value = int(width_value)
                                if width_value <= 0:
                                    width_value = 1
                                elif width_value > 14:
                                    width_value = 14
                                options['size_of_grid'][0] = width_value
                            set_square_value()

                        #Retrieve and display number of rows input
                        elif (x >= sog2_co[0] and x <= sog2_co[1]) and (y
>= sog2_co[2] and y <= sog2_co[3]):
                            coords = (1210, 95, 1240, 115)
                            size = (30, 20)
                            size_of_grid2_input =
text_input.text_input_display(screen, coords, size)
                            height_value =
size_of_grid2_input.completed_text
                            if len(height_value) != 0:
                                height_value = int(height_value)
                                if height_value <= 0:
                                    height_value = 1
                                elif height_value > 10:
                                    height_value = 10
                                options['size_of_grid'][1] = height_value
                            set_square_value()

                        #Detect and update currently selected colour
                        elif (x >= sp_co[0] and x <= sp_co[1]) and (y >=
sp_co[2] and y <= sp_co[3]):
                            options['selected_colour'] = red
                        elif (x >= pt_co[0] and x <= pt_co[1]) and (y >=
pt_co[2] and y <= pt_co[3]):
                            options['selected_colour'] = white
                        elif (x >= it_co[0] and x <= it_co[1]) and (y >=
it_co[2] and y <= it_co[3]):
                            options['selected_colour'] = grey
                        elif (x >= dp_co[0] and x <= dp_co[1]) and (y >=
dp_co[2] and y <= dp_co[3]):
                            options['selected_colour'] = green
                        elif (x >= dl_co[0] and x <= dl_co[1]) and (y >=
dl_co[2] and y <= dl_co[3]):
                            options['selected_colour'] = black

                    #If there are lines on the screen, clear lines button
is displayed, and this button is clicked:
                    if len(list_of_lines) != 0 and (x >= cl_co[0] and x <=
cl_co[1]) and (y >= cl_co[2] and y <= cl_co[3]):
                        #Reset variables holding characteristics of lines,
lines are cleared from the screen
                        list_of_lines = []
                        list_of_circles = []
                        line_coordinates = []
```

```
                        options['lines'] = False
                        start_cell, end_cell = "", ""

                    #If a location within the grid has been clicked
                    if (x >= grid_co[0] and x <= grid_co[1]) and (y >=
grid_co[2] and y <= grid_co[3]):
                        #Retrieve the size of each cell in pixels
                        size_of_square = options['size_of_square']
                        #x, y position relative to the start of the grid
                        temp_x, temp_y = x - 30, y - 30

                        #If the click wasn't on the border of a cell (was
within a a cell)
                        if (temp_x % size_of_square) != 0 and (temp_y %
size_of_square) != 0:
                            #Find the column and row that the click was
registered in
                            column = temp_x / size_of_square
                            row = temp_y / size_of_square
                            #A list of all the 8 cells surrounding the cell
clicked, starting from the right and going clockwise
                            surrounding_cells = [[column+1, row],
[column+1, row+1], [column, row+1], [column-1, row+1], [column-1, row],
[column-1, row-1], [column, row-1], [column+1, row-1]]

                            #If the current colour/ property to be assigned
to a cell is passable terrain
                            if options['selected_colour'] == green or
options['selected_colour'] == red or options['selected_colour'] == white:
                                #Turn all the cells around the cell clicked
grey (since two passable terrain cells can't be adjacent)
                                for i in surrounding_cells:
                                    #Checks that cell isn't outside the
grid so an error doesn't occur
                                    if (i[0] >= 0 and i[0] <
options['size_of_grid'][0]) and (i[1] >= 0 and i[1] <
options['size_of_grid'][1]):
                                        grid_references[i[0]][i[1]][2] =
grey

                                #If the current colour/ property is the
start or destination cell then remove the old start/ destination cell from
the grid
                                if options['selected_colour'] == green or
options['selected_colour'] == red:
                                    for i in
range(options['size_of_grid'][0]):
                                        for j in
range(options['size_of_grid'][1]):
                                            if grid_references[i][j][2] ==
green and options['selected_colour'] == green:
                                                grid_references[i][j][2] =
grey
                                            elif grid_references[i][j][2]
== red and options['selected_colour'] == red:
                                                grid_references[i][j][2] =
grey

                            #If the selected colour is black - A line is
being drawn
                            elif options['selected_colour'] == black:
```

```
                                #If the cell being clicked is passable
terrain - Validates you aren't trying to draw a line to an impassable
terrain cell
                                if grid_references[column][row][2] == green
or grid_references[column][row][2] == red or
grid_references[column][row][2] == white:
                                    #If the click is the start of a line
                                    if options['start_of_line'] == True:
                                        #Assign the column and row of the
first cell of the line to a variable
                                        start_grid = [column, row]
                                        #Assign the colour/ property of a
cell to a variable so it can be restored later
                                        start_colour =
grid_references[column][row][2]

                                        #If the start colour is red, shade
the cell light red to show it is the start of a line
                                        if start_colour == red:
                                            #Start cell is filled in as
validation
                                            start_cell = [column, row]
                                            grid_references[column][row][2]
= light_red
                                        #If the start colour is red, shade
the cell light red to show it is the start of a line
                                        elif start_colour == green:
                                            #End cell is filled in as
validation

                                            end_cell = [column, row]
                                            grid_references[column][row][2]
= light_green
                                        #If the cell is white, shade it
light grey
                                        else:
                                            grid_references[column][row][2]
= light_grey


                                        #Note the start of the line for
drawing it
                                        start_of_line =
grid_references[column][row]
                                        #Get the coordinates of the top
left of the cell
                                        startx, starty = start_of_line[0],
start_of_line[1]
                                        #Get the coordinates of the centre
of the cell, relative to the screen
                                        startx += 29 +
(options['size_of_square']/2)
                                        starty += 29 +
(options['size_of_square']/2)
                                        #Start of line is now false, so
that the next click draws the line
                                        options['start_of_line'] = False


                                    #If selecting the end coordinate of the
line
                                    elif options['start_of_line'] == False:
                                        #Validation
```

```
                                        if grid_references[column][row][2]
== red:
                                            start_cell = [column, row]
                                        elif
grid_references[column][row][2] == green:
                                            end_cell = [column, row]

                                        #Returning the starting cell back
to it's original colour

grid_references[start_grid[0]][start_grid[1]][2] = start_colour
                                        #Assigning appropriate coordinates
of the end cell to variables
                                        end_of_line =
grid_references[column][row]
                                        endx, endy = end_of_line[0],
end_of_line[1]
                                        endx += 29 +
(options['size_of_square']/2)
                                        endy += 29 +
(options['size_of_square']/2)

                                        #If a line has been drawn
                                        if len(list_of_lines) != 0:
                                            #The coordinates of the line
being drawn - Take the start co-ordinates of this line as A and end
coordinates as B, so line is AB [startx, starty, endx, endy]
                                            line1 = [start_grid[0],
start_grid[1], column, row]
                                            #The reverse of a line so
[endx, endy, startx, starty]
                                            reverse_line1 = [line1[2],
line1[3], line1[0], line1[1]]
                                            #Validation check
                                            intersect = False

                                            #Loop through all the lines and
check if they intersect with the line being drawn
                                            for i in line_coordinates:
                                                #Validation variables for
each line
                                                intersect1, intersect2 =
False, False
                                                #Assigning the coordinates
of each line in line coordinates to a variable that is easier to understand
- Take the start co-ordinates of this line as C and end coordinates as D,
so line is CD
                                                line2 = i

                                                #Initial validation -
Checking the line being drawn isn't the same line, or the same line being
drawn the other way, as a line already on the grid - Removing duplicate
lines
                                                if line1 == line2 or
reverse_line1 == line2:
                                                    #Intersect variable for
validation is True, the line won't be drawn
                                                    intersect = True

                                                #With the line being drawn
as AB and each line in the list of lines being CD, we can use the following
```

```
formula to check if they intersect - Ax would be the x value for A, Ay
would be the y coordinate
                                                    else:
                                                        #(Dx - Cx)(Ay - Dy) -
(Dy - Cy)(Ax - Dx)
                                                        x = (line2[2] -
line2[0])*(line1[1] - line2[3]) - (line2[3] - line2[1])*(line1[0] -
line2[2])
                                                        #(Dx - Cx)(By - Fy) -
(Dy - Cy)(Bx - Dx)
                                                        y = (line2[2] -
line2[0])*(line1[3] - line2[3]) - (line2[3] - line2[1])*(line1[2] -
line2[2])

                                                        #If x and y are
opposite signs, one is positive and one is negative, then one coordinate of
the line being drawn is within the boundary of another line - We then
perform the check with another coordinate
                                                        if x > 0 and y < 0:
                                                            intersect1 = True
                                                        elif x < 0 and y > 0:
                                                            intersect1 = True

                                                        #(Bx - Ax)(Cy - By) -
(By - Ay)(Cx - Bx)
                                                        a = (line1[2] -
line1[0])*(line2[1] - line1[3]) - (line1[3] - line1[1])*(line2[0] -
line1[2])
                                                        #(Bx - Ax)(Dy - By) -
(By - Ay)(Dx - Bx)
                                                        b = (line1[2] -
line1[0])*(line2[3] - line1[3]) - (line1[3] - line1[1])*(line2[2] -
line1[2])

                                                        #Checking if the other
coordinate of the line being drawn is within the boundary of another line
                                                        if a > 0 and b < 0:
                                                            intersect2 = True
                                                        elif a < 0 and b > 0:
                                                            intersect2 = True

                                                        #If both coordinates
are within the boundaries of another line, that means it intersects
                                                        if intersect1 == True
and intersect2 == True:
                                                            #The intersect
variable is assigned to True, so the line will not be drawn
                                                            intersect = True

                                                #If this is the first line being
drawn (can't intersect with another line) or if it isn't the first line but
is valid, then draw the line
                                                if len(list_of_lines) == 0 or
intersect == False:
                                                    #Finding the length of the line
[dx, dy]
                                                    size = [endx-startx, endy-
starty]
                                                    #Getting the coordinates of the
middle of the line, then taking away half the size of the box that contains
the number for the length of the line - So the box is drawn in the centre
```

```
                                        middlex = startx + (size[0]/2)
                                        middlex -= 13
                                        middley = starty + (size[1]/2)
                                        middley -= 13
                                        #Use the Pythagoras theorem to
work out the distance between the two lines
                                        pythag = ((column-
start_grid[0])**2 + (row-start_grid[1])**2)**0.5
                                        #Rounding the weight of the
line to two significant figures
                                        weight = round(pythag, 2-
int(floor(log10(pythag)))-1)

                                        #If the line ends in .0
                                        if str(weight-int(weight))[1:]
== ".0":
                                            #Remove the .0 so that it
just displays the number
                                            weight = int(weight)

                                        #Append the coordinates of
start and end coordinates of the line to the list, along with the
coordinates where the box will be drawn and the weight - The box is drawn
later with the line - This is used to draw the lines
                                        list_of_lines.append([startx,
starty, endx, endy, middlex, middley, weight])
                                        #Append the start and end cell
reference, instead of the pixel coordinate, to a separate list - This is
used for performing Dijkstra's Algorithm

line_coordinates.append([start_grid[0], start_grid[1], column, row,
weight])

                                        #If either the start or the end
of a line is on a cell that hasn't had a line drawn to it yet, append it to
the list of circles, since these are drawn before the lines so that they
always appear below them
                                        if [startx, starty] not in
list_of_circles:

list_of_circles.append([startx, starty])
                                        if [endx, endy] not in
list_of_circles:

list_of_circles.append([endx, endy])

                                    #Once a has been drawn, or failed
to be drawn, the start of line option is now True again so the next click
starts another line
                                    options['start_of_line'] = True
                                    #Option to show lines have been
drawn - Recolours options to show they are disabled
                                    options['lines'] = True

                                #If the selected colour isn't a line - Then
update the cell with the colour selected
                                if options['selected_colour'] != black:
                                    grid_references[column][row][2] =
options['selected_colour']
```

```
                    #If the user clicks outside the grid, the start of the
line is reset
                    else:
                        options['start_of_line'] = True
                        if start_grid != "":

grid_references[start_grid[0]][start_grid[1]][2] = start_colour

                    #Redraw the options screen with any changes
                    options_screen = function1_screen()
                    options_screen.draw_options_screen()
                    coordinate_list = options_screen.coordinate_list

                #If the user clicks and the options aren't displayed, other
buttons are available
                elif options['show_options'] == False:
                    #If the user clicks the home button
                    if (x >= home_co[0] and x <= home_co[1]) and (y >=
home_co[2] and y <= home_co[3]):
                        #Erase all variables associated with the grid, so
that not of it carries over if the user decides to start a new grid
                        grid_references = []
                        list_of_lines = []
                        list_of_circles = []
                        line_coordinates = []
                        options['lines'] = False
                        options['solve'] = False
                        options['show_options_arrow'] = True
                        #Returns the user to the choose function page again
                        choose_function()

                    #If either the start or destination cell doesn't have a
line attached to it then the solve button shouldn't be displayed
                    if start_cell == "" or end_cell == "":
                        options['show_solve_button'] = False
                    else:
                        options['show_solve_button'] = True

                    #If the solve button is available, validation has
checked it should be displayed
                    if options['show_solve_button'] == True:
                        #Retrieve the solve button co-ordinates, this is
retrieved now so the user cannot click the button when it isn't displayed
                        sb_co = coordinate_list[1]
                        #If the user clicks the solve button
                        if (x >= sb_co[0] and x <= sb_co[1]) and (y >=
sb_co[2] and y <= sb_co[3]):
                            #If solve is False, then the graph is ready to
be solved
                            if options['solve'] == False:
                                #Solve option turns true
                                options['solve'] = True
                                #Remove options arrow when the graph is
solved
                                options['show_options_arrow'] = False
                                #Run function2 of the perform_dijkstras
file, to retrieve the final route - Is displayed in another class
                                final_route =
perform_dijkstras.function2(line_coordinates, start_cell, end_cell)
```

```
                            #If solve is true, then the graph has been
solved, so clicking this button again returns the user back to the editing
screen
                            elif options['solve'] == True:
                                options['solve'] = False
                                options['show_options_arrow'] = True

                    #Redraw the screen to reflect any changes made
                    no_options_screen = function1_screen()
                    no_options_screen.draw_no_options_screen()
                    coordinate_list = no_options_screen.coordinate_list

        #Update the display
        pygame.display.update()

#Creating a class for creation of buttons and text boxes
class box(object):
    #This runs automatically, the default place holder variables that are
then changed by a module or by editing the instance
    def __init__(self):
        #Fill colour of the box
        self.fill_colour = white
        #Location, in pixels, from the top left of the screen to the top
left of the box
        self.location = (0, 0)
        #Size of box in pixels
        self.size = (0, 0)
        #Colour of the outline of the box
        self.outline_colour = black
        #Size of the outline of the box
        self.size_outline = 1
        #Font size of text within the box
        self.font_size = 18
        #Text input, if any
        self.text_input = ""
        #Location of the text within the box, usually defined by a module
but can be custom defined
        self.text_location = (0, 0)
        #If the text is bold
        self.bold = False

    #Module for the standard button used in throughout the menu
    def default_button(self):
        self.size = (150, 100)
        self.font_size = 20

    #Module for standard text box, used in the main menu
    def default_text_box(self):
        self.location = (200, 450)
        self.size = (880, 150)
        self.font_size = 16

    #Module for standard button featured on the options screen at the side
of the grid
    def default_options_buttons(self):
        self.size = (20, 20)
        self.font_size = 16
        self.size_outline = 2

    #Module that draws the box on screen once the size and location has
been defined
```

```
    def draw_box(self):
        #The outline of the box is created by drawing a larger box behind
the first
        outline_location = (self.location[0] - self.size_outline,
self.location[1] - self.size_outline)
        outline_size = (self.size[0] + (self.size_outline*2), self.size[1]
+ (self.size_outline*2))

        #Drawing first the outline box, then the actual box on top to
create the button/ text box
        screen.lock()
        pygame.draw.rect(screen, self.outline_colour, (outline_location,
outline_size))
        pygame.draw.rect(screen, self.fill_colour, (self.location,
self.size))
        screen.unlock()

    #Preparing the necessary variables for displaying text on screen, this
is separate in case the text needs to be centralised before displayed
    def prep(self):
        self.x_position, self.y_position = self.text_location
        self.font = pygame.font.SysFont("calibri", self.font_size,
self.bold)
        self.text = self.font.render(self.text_input, 1, black)
#rendering the text on to the screen

    #Getting the start and end coordinates of the box
    def get_coords(self):
        x1 = self.location[0]
        x2 = self.location[0] + self.size[0]
        y1 = self.location[1]
        y2 = self.location[1] + self.size[1]
        return x1, x2, y1, y2

    #Runs two modules to centralise the text in both axes
    def centralise_text(self):
        self.centralise_x(), self.centralise_y()

    #Centralises the text in the x axis
    def centralise_x(self):
        #Gets the middle coordinate of the box
        self.x_position = self.location[0] + (self.size[0]/2)
        #Subtracts half the width of the text
        self.x_position -= ((self.text.get_width()+1)/2)

    #Centralises the text in the y axis
    def centralise_y(self):
        self.y_position = self.location[1] + (self.size[1]/2)
        self.y_position  -= (((self.text.get_height()+1)*(3/4.))/2)

    #Display the final text on the screen
    def display_text(self):
        screen.blit(self.text, (self.x_position, self.y_position))

#Class containing the necessary modules to display the main function screen
of the program
class function1_screen(object):
    #Displays the workspace area using the box class, the white box that
the grid and options are displayed on top of
    def __init__(self):
        workspace = box()
```

```
        workspace.location = (10, 10)
        workspace.size = (1260, 700)
        workspace.draw_box()
        #Resets coordinate list each time, since it is used for both
functions
        self.coordinate_list = []

    #Displays the home button using the box class
    def home_button(self):
        home = box()
        home.default_button()
        home.location = (1090, 90)
        home.size = (150, 30)
        home.font_size = 20 #1240, 65
        home.text_input = "Main Menu"
        home.draw_box(), home.prep(), home.centralise_text(),
home.display_text()
        self.home_button_coordinates = home.get_coords()

    #Creates all the buttons necessary for the options bar at the side of
the screen during the main program
    def options_box_create(self):
        #Creating the options bar
        options_box = box()
        options_box.location = (1000, 30)
        options_box.size = (250, 660)
        options_box.size_outline = 2
        options_box.draw_box()

        #Displaying options text at top of options bar
        font = pygame.font.SysFont("calibri", 18)
        screen.blit(font.render("Options:", 1, black), (1010, 45))

        #Creating inverted options arrow to close the options bar
        self.arrow_box_create()
        pygame.draw.polygon(screen, black, [(1220, 45), (1221, 45), (1233,
52), (1233, 53), (1221, 60), (1220, 60)])

        #Displays the first size of grid input along with the text "Size of
Grid:"
        screen.blit(pygame.font.SysFont("calibri", 16).render("Size of
Grid:", 1, black), (1010, 100))
        size_of_grid = box()
        size_of_grid.default_options_buttons()
        size_of_grid.location = (1150, 95)
        size_of_grid.size = (30, 20)
        size_of_grid.text_input = str(options['size_of_grid'][0])
        if options['lines'] == True:
            size_of_grid.fill_colour = light_grey
        size_of_grid.draw_box(), size_of_grid.prep(),
size_of_grid.centralise_text(), size_of_grid.display_text()
        sog_coords = size_of_grid.get_coords()

        #Displays the second size of grid input along with the "x" in
between the two inputs
        screen.blit(pygame.font.SysFont("calibri", 16).render("x", 1,
black), (1190, 100))
        size_of_grid2 = box()
        size_of_grid2.default_options_buttons()
        size_of_grid2.location = (1210, 95)
        size_of_grid2.size = (30, 20)
```

```
        size_of_grid2.text_input = str(options['size_of_grid'][1])
        if options['lines'] == True:
            size_of_grid2.fill_colour = light_grey
        size_of_grid2.draw_box(), size_of_grid2.prep(),
size_of_grid2.centralise_text(), size_of_grid2.display_text()
        sog2_coords = size_of_grid2.get_coords()

        #Displays the text "Max Size of Grid: a x b" depending on the
function (40 x 30 for Grid function, 14 x 10 for Vertices function)
        font = pygame.font.SysFont("calibri", 14)
        max_size1 = font.render("Max Size of Grid:", 1, black)
        screen.blit(max_size1, (1010, 125))
        if options['function'] == 1:
            max_size2 = font.render("40", 1, black)
        elif options['function'] == 2:
            max_size2 = font.render("14", 1, black)
        screen.blit(max_size2, (1157, 125))
        max_size3 = font.render("x", 1, black)
        screen.blit(max_size3, (1190, 125))
        if options['function'] == 1:
            max_size4 = font.render("30", 1, black)
        elif options['function'] == 2:
            max_size4 = font.render("10", 1, black)
        screen.blit(max_size4, (1217, 125))

        #Creates the start position option box
        start_position = box()
        start_position.default_options_buttons()
        start_position.location = (1220, 160)
        start_position.text_location = (1010, 165)
        start_position.text_input = "Start Position:"
        if options['lines'] == False:
            start_position.fill_colour = red
        else:
            #For when the options are disabled
            start_position.fill_colour = light_grey
        start_position.draw_box(), start_position.prep(),
start_position.display_text()
        sp_coords = start_position.get_coords()

        #Creates the passable terrain option
        passable_terrain = box()
        passable_terrain.default_options_buttons()
        passable_terrain.location = (1220, 200)
        passable_terrain.text_location = (1010, 205)
        passable_terrain.text_input = "Passable Terrain:"
        if options['lines'] == True:
            passable_terrain.fill_colour = light_grey
        passable_terrain.draw_box(), passable_terrain.prep(),
passable_terrain.display_text()
        pt_coords = passable_terrain.get_coords()

        #If the Grid function is working then there is a slow terrain
option
        if options['function'] == 1:
            slow_terrain = box()
            slow_terrain.default_options_buttons()
            slow_terrain.location = (1220, 240)
            slow_terrain.text_location = (1010, 245)
            slow_terrain.text_input = "Slow Terrain:"
            if options['lines'] == False:
```

```
                slow_terrain.fill_colour = brown
            else:
                slow_terrain.fill_colour = light_grey
            slow_terrain.draw_box(), slow_terrain.prep(),
slow_terrain.display_text()
            st_coords = slow_terrain.get_coords()

        #Impassable terrain option
        impassable_terrain = box()
        impassable_terrain.default_options_buttons()
        impassable_terrain.location = (1220, 280)
        impassable_terrain.text_location = (1010, 285)
        if options['function'] == 2:
            #Since there is no slow terrain all other options have to be
moved up
            impassable_terrain.location = (1220, 240)
            impassable_terrain.text_location = (1010, 245)
        impassable_terrain.text_input = "Impassable Terrain:"
        if options['lines'] == False:
            impassable_terrain.fill_colour = grey
        else:
            impassable_terrain.fill_colour = light_grey
        impassable_terrain.draw_box(), impassable_terrain.prep(),
impassable_terrain.display_text()
        it_coords = impassable_terrain.get_coords()

        #Destination position option
        destination_position = box()
        destination_position.default_options_buttons()
        destination_position.location = (1220, 320)
        destination_position.text_location = (1010, 325)
        if options['function'] == 2:
            destination_position.location = (1220, 280)
            destination_position.text_location = (1010, 285)
        destination_position.text_input = "Destination Position:"
        if options['lines'] == False:
            destination_position.fill_colour = green
        else:
            destination_position.fill_colour = light_grey
        destination_position.draw_box(), destination_position.prep(),
destination_position.display_text()
        dp_coords = destination_position.get_coords()

        #If the Vertices function is running then the draw lines option is
displayed at the bottom of the options
        if options['function'] == 2:
            draw_line = box()
            draw_line.default_options_buttons()
            draw_line.location = (1220, 320)
            draw_line.text_location = (1010, 325)
            draw_line.text_input = "Draw Line"
            draw_line.fill_colour = black
            draw_line.draw_box(), draw_line.prep(),
draw_line.display_text()
            dl_coords = draw_line.get_coords()

        #Displays the currently selected colour so teh user knows what
property they will assign to a cell if they click one
        selected_colour = box()
        selected_colour.default_options_buttons()
        selected_colour.location = (1220, 370)
```

```
        selected_colour.text_location = (1010, 375)
        selected_colour.text_input = "Selected Colour:"
        selected_colour.font_size = 16
        selected_colour.bold = True
        if options['lines'] == False:
            selected_colour.fill_colour = options['selected_colour']
        else:
            selected_colour.fill_colour = black
        selected_colour.draw_box(), selected_colour.prep(),
selected_colour.display_text()

        #If the Vertices function is running and a line has been drawn on
the screen then the Clear Lines button needs to be displayed
        if options['function'] == 2 and len(list_of_lines) != 0:
            clear_lines = box()
            clear_lines.location = (1050, 600)
            clear_lines.size = (150, 40)
            clear_lines.text_input = "Clear current lines"
            clear_lines.font_size = 18
            clear_lines.draw_box(), clear_lines.prep(),
clear_lines.centralise_text(), clear_lines.display_text()
            cl_coords = clear_lines.get_coords()

        #Adds the coordinates of each button the coordinate list (extend is
the same as append but for multiple items
        self.coordinate_list.extend([sog_coords, sog2_coords, sp_coords,
pt_coords, it_coords, dp_coords])
        #If the Grid function is running it appends the slow terrain
coordinates at the end
        if options['function'] == 1:
            self.coordinate_list.append(st_coords)
        #If the Vertices function is running it appends the draw line
coordinates at the end along with the clear lines if there are lines drawn
        elif options['function'] == 2:
            self.coordinate_list.append(dl_coords)
            if len(list_of_lines) != 0:
                self.coordinate_list.append(cl_coords)

    #Displays the arrow in the top right corner to open/ close the option
bar
    def arrow_box_create(self):
        arrow_box = box()
        arrow_box.location = (1215, 40)
        arrow_box.size = (25, 25)
        arrow_box.draw_box()
        self.arrow_box_coordinates = arrow_box.get_coords()

    #Creates the solve button
    def solve_box_create(self):
        solve_button = box()
        solve_button.default_button()
        solve_button.location = (1025, 600)
        solve_button.size = (200, 50)
        solve_button.font_size = 20
        if options['solve'] == False:
            solve_button.text_input = "Solve"
        #If the graph is solved the button returns the user to the editing
screen, so the text is changed
        elif options['solve'] == True:
            solve_button.text_input = "Return"
```

```
        solve_button.draw_box(), solve_button.prep(),
solve_button.centralise_text(), solve_button.display_text()
        self.solve_button_coordinates = solve_button.get_coords()

    #This module performs a basic validation check, if the grid contains
the start and destination cells required to solve the grid
    def grid_validity(self):
        green_valid, red_valid = False, False
        #Loops through all the cells in the grid
        for i in range(options['size_of_grid'][0]):
            for j in range(options['size_of_grid'][1]):
                #If the end cell is present updates the variable to show it
is
                if grid_references[i][j][2] == green:
                    green_valid = True
                #If the start cell is present updates the variable to show
it is
                elif grid_references[i][j][2] == red:
                    red_valid = True

        #If the Grid function is running then validate the grid (the
Vertices function requires extra validation to detect if there are lines
drawn to/ from the start and destination coordinates)
        if options['function'] == 1:
            #If both the start and destination cells are present, the Solve
button is displayed
            if green_valid == True and red_valid == True:
                options['show_solve_button'] = True
            else:
                options['show_solve_button'] = False

    #Module that runs other modules necessary to create the main program
screen (such as creating the grid references)
    def draw_initial_screen(self):
        #Instance of the display_grid class below
        grid = display_grid()
        #Initialises the grid - Creates lists and draws the grid
        grid.initialise_grid()

        #Runs the arrow_box_create module to display the arrow box
        self.arrow_box_create()
        #Draws the initial arrow
        pygame.draw.polygon(screen, black, [(1235, 45), (1234, 45), (1222,
52), (1222, 53), (1234, 60), (1235, 60)])

        #Appends the coordinates of the outline of the grid to the
coordinate list
        self.coordinate_list.append(grid.grid_coordinates)
        #Check if the grid is valid
        self.grid_validity()

        #If the grid is valid and the solve button can be pressed then
display the solve button and append the coordinates
        if options['show_solve_button'] == True:
            self.solve_box_create()
            self.coordinate_list.append(self.solve_button_coordinates)

        #Create the home button
        self.home_button()
        self.coordinate_list.append(self.home_button_coordinates)
```

```
    #Draw the screen with the options bar displayed
    def draw_options_screen(self):
        #Instance of display_grid class
        grid = display_grid()
        #Update the grid - Separate from initialise grid since it doesn't
have to create the grid coordinates from scratch
        grid.update_grid()
        #Append the coordinates to the list
        self.coordinate_list.append(grid.grid_coordinates)
        #Run the options_box_create module
        self.options_box_create()

    #Draw the screen without the options bar displayed
    def draw_no_options_screen(self):
        #Displaying grid and home button
        grid = display_grid()
        grid.update_grid()
        self.coordinate_list.append(grid.grid_coordinates)
        self.grid_validity()
        self.home_button()

        #If the options arrow should be displayed then display it
        if options['show_options_arrow'] == True:
            self.arrow_box_create()
            pygame.draw.polygon(screen, black, [(1235, 45), (1234, 45),
(1222, 52), (1222, 53), (1234, 60), (1235, 60)])

        #If the solve button should be displayed then display it
        if options['show_solve_button'] == True:
            self.solve_box_create()
            self.coordinate_list.append(self.solve_button_coordinates)

        #If the graph has selected to be solved
        if options['solve'] == True:
            #If it Grid function
            if options['function'] == 1:
                #Retrieve the shortest route by running the main function
of the perform_dijkstras file
                finished_route =
perform_dijkstras.main(options['size_of_grid'], grid_references)
                #Run the draw_shortest_route module of the grid class to
display the shortest route
                grid.draw_shortest_route(finished_route)

        #Display the home button
        self.home_button()
        self.coordinate_list.append(self.home_button_coordinates)

#Class for everything to do with displaying the grid
class display_grid(object):
    #Runs every time this class is called
    def __init__(self):
        #Using the global grid_references variable instead of creating a
new one
        global grid_references
        #Retrieving necessary information from options dictionary
        self.size_of_grid = options['size_of_grid']
        self.size_of_square = options['size_of_square']

        #Creating the outline of the grid
        grid_outline = box()
```

```
        grid_outline.location = (30, 30)
        grid_outline.size = ((self.size_of_grid[0]*self.size_of_square) +
1, (self.size_of_grid[1]*self.size_of_square) + 1)
        grid_outline.size_outline = 1
        grid_outline.fill_colour = black
        grid_outline.draw_box()
        self.grid_coordinates = grid_outline.get_coords()

    #Module to create and draw the grid
    def initialise_grid(self):
        self.create_grid_references()
        self.draw_grid()

    #Module to create list of all grid references
    def create_grid_references(self):
        global grid_references
        #Loops through the number of columns that should be displayed
        for i in range(self.size_of_grid[0]):
            #The x coordinate, in pixels, of the start of each cell in that
column
            self.x = (i * self.size_of_square) + 1
            #Append an empty list to the grid_references to hold all of the
rows in that column
            grid_references.append([])

            #Loops through all the rows in that column
            for j in range(self.size_of_grid[1]):
                #Creates the y coordinate of each cell in that column
                self.y = (j * self.size_of_square) + 1
                #If the Grid function is running then all the cells are
default white/ passable terrain
                if options['function'] == 1:
                    grid_references[i].append([self.x, self.y, white])
                #If the Vertices function is running then all the cells are
default grey/ impassable terrain
                elif options['function'] == 2:
                    grid_references[i].append([self.x, self.y, grey])

        #Assign the top left cell to be the start cell and the bottom right
cell to be the destination cell as default placeholders
        grid_references[0][0][2], grid_references[self.size_of_grid[0]-
1][self.size_of_grid[1]-1][2] = red, green

    #Draw the grid from the grid references
    def draw_grid(self):
        global grid_references#
        #Loops through all the cells
        for i in range(self.size_of_grid[0]):
            for j in range(self.size_of_grid[1]):
                #Assigns the x and y coordinate of the top left of each
cell to variables
                x_coordinate = grid_references[i][j][0]
                y_coordinate = grid_references[i][j][1]
                #Retrieves the colour of the cell from the grid references
to be displayed
                colour_of_cell = grid_references[i][j][2]
                #Draw each cell on the grid (Where it's being drawn,
colour, ((x coordinate of start of cell, y coordinate of start of cell),
(size of each cell))
```

```
                pygame.draw.rect(screen, colour_of_cell, ((x_coordinate +
30, y_coordinate + 30), ((self.size_of_square-1), (self.size_of_square-
1))))

        #If the Vertices function is running
        if options['function'] == 2:
            #Draw a circle for each circle in the list - To show the start
and end of each line more clearly
            for i in list_of_circles:
                pygame.draw.circle(screen, light_blue, (i[0], i[1]), 7)

            #Variables for drawing the lines on screen
            count = 0
            finished_route = []

            #Loops through each line drawn on screen
            for i in list_of_lines:
                #If the graph has been solved and that line is in the final
route retrieved from the perform_dijkstras file, then another list is
created with the x and y coordinates or each line, instead of the column
and row number
                if options['solve'] == True and line_coordinates[count] in
final_route:
                    finished_route.append(i)
                #If the line is not in the finished route then draw the
line, including the text box in the middle displaying the weight of the
line
                else:
                    pygame.draw.line(screen, black, (i[0]-1, i[1]-1),
(i[2]-1, i[3]-1), 2)
                    pygame.draw.rect(screen, black, ((i[4]-1, i[5]-1), (27,
27)))
                    pygame.draw.rect(screen, white, ((i[4], i[5]), (25,
25)))
                    font = pygame.font.SysFont("calibri", 16)
                    text = font.render(str(i[6]), 1, black)
                    middlex = i[4] + 12
                    middlex -= ((text.get_width()+1)/2)
                    middley = i[5] + 12
                    middley -= (((text.get_height()+1)*(3/4.))/2)
                    screen.blit(text, (middlex, middley))

                #Count variable for looping through the line_coordinates
                count += 1

            #For each line in the finished route, draw the line but in blue
- This happens after so that the finished route is always displayed on top
of other lines
            for i in finished_route:
                pygame.draw.line(screen, light_blue, (i[0]-1, i[1]-1),
(i[2]-1, i[3]-1), 3)
                pygame.draw.rect(screen, light_blue, ((i[4]-1, i[5]-1),
(27, 27)))
                pygame.draw.rect(screen, white, ((i[4], i[5]), (25, 25)))
                font = pygame.font.SysFont("calibri", 16)
                text = font.render(str(i[6]), 1, black)
                middlex = i[4] + 12
                middlex -= ((text.get_width()+1)/2)
                middley = i[5] + 12
                middley -= (((text.get_height()+1)*(3/4.))/2)
                screen.blit(text, (middlex, middley))
```

```
    #Module for updating the grid - Since the grid references don't need to
be created again
    def update_grid(self):
        global grid_references
        #Store the original grid references and grid size before any
changes
        old_grid_references = grid_references
        old_size_of_grid = [len(old_grid_references),
len(old_grid_references[0])]
        #Retrieve the new size of the grid from the options
        new_size_of_grid = options['size_of_grid']

        #If the grid size has been changed, then it recreates the grid
references, but keeping any old cells
        if old_size_of_grid != new_size_of_grid:
            grid_references = []
            #Loops through all the cells
            for i in range(new_size_of_grid[0]):
                x = (i * self.size_of_square) + 1
                grid_references.append([])
                for j in range (new_size_of_grid[1]):
                    y = (j * self.size_of_square) + 1
                    #This is where it's different, if the cell is within
the size of the old grid
                    if i < old_size_of_grid[0] and j < old_size_of_grid[1]:
                        #It retrieves the property of that cell from the
old grid_references
                        colour = old_grid_references[i][j][2]

                    #Otherwise it fills the cell in with it's default
property depending on the function
                    else:
                        if options['function'] == 1:
                            colour = white
                        elif options['function'] == 2:
                            colour = grey

                    #Updating the new grid references with the coordinate
and colour of each cell
                    grid_references[i].append([x, y, colour])

        #Draw the updated grid
        self.draw_grid()

    #Module for drawing the shortest route - For the Grid function
    def draw_shortest_route(self, finished_route):
        #For each cell in the finished route
        for i in range(len(finished_route)):
            #Retrieve the coordinate of the cell and find the middle of the
cell
            current_cell =
grid_references[finished_route[i][0]][finished_route[i][1]]
            x = current_cell[0]
            y = current_cell[1]
            x += 29 + (self.size_of_square/2)
            y += 29 + (self.size_of_square/2)

            #Replace the column/ row number with the x and y coordinate of
the centre of each cell in the finished route
            finished_route[i] = (x, y)
```

```
        #Loops through the finished route and draws a line from the centre
of each cell to the next
        for i in range(len(finished_route)-1):
            pygame.draw.line(screen, black, finished_route[i],
finished_route[i+1], 2)

#If the program is being run by itself, not through another file, then
perform the main_menu function
if __name__ == "__main__":
    main_menu()
```

Text Input

Here is the fully annotated code for the text input file used in my program. This section of code controls any areas of my program where the user can input data, which are predominantly the inputs for the size of the grid.

```
#Importing modules
import pygame, sys
#RGB value for black
black = (0, 0, 0)

#Class containing the code for the retrieving inputs and displaying them on
screen
class text_input_display():
    def __init__(self, screen, co_ords, size):
        #Assigning the start and end coordinates of the box it is being
displayed in
        x, y, endx, endy = co_ords
        #Place holder variables
        empty_string = ""
        self.completed_text = ""
        current_list = []
        done = False

        #Whilst done doesn't equal False, run through loop. done = False
until stated otherwise in the code
        while done == False:
            #Coordinates of mouse
            mousex, mousey = pygame.mouse.get_pos()
            #Loops through all events in the event loop (button clicks,
entered keyboard characters etc.)
            for event in pygame.event.get():
                #If the current event is QUIT (the X in the top right of
the window)
                if event.type == pygame.QUIT:
                    #Quits the program
                    pygame.quit()
                    sys.exit()

                #If the mouse is clicked and it's outside the input box:
                elif (event.type == pygame.MOUSEBUTTONUP) and ((mousex < x
or mousex > endx) or (mousey < y or mousey > endy)):
                    #Append each of the character of the list to the
variable "completed_text"
                    self.completed_text =
self.completed_text.join(current_list)
                    #done = True, so the while loop closes and moves onto
the code after it (exiting the program)
                    done = True

                #If the current event is a keyboard button being pressed
down
                elif event.type == pygame.KEYDOWN:
                    #If the key of the current event is the backspace being
pressed down
                    if event.key == pygame.K_BACKSPACE:
                        #The current list is now 1 less in length of what
it was before (removes the last character of the list)
                        current_list = current_list[0:-1]
                    #If the current key is the enter key
```

```
                    elif event.key == 13 or event.key == 271:
                        #Append each character of the list to the variable
"completed_text"
                        self.completed_text =
self.completed_text.join(current_list)
                        #Done is now True, so the loop ends
                        done = True

                    #If the current key is a number being pressed down and
the length of the current list is less than 2, there is room for another
character, then convert it to a number
                    elif ((event.key >= 48 and event.key <= 57) or
(event.key >= 256 and event.key <= 265)) and len(current_list) < 2:
                        #This changes numbers entered on the number pad to
be the same as the ones at the top of the keyboard, so I can easily convert
them
                        if event.key >= 256:
                            key = event.key - 208
                        else:
                            key = event.key

                        #Convert the key into a character and append it to
the list
                        current_list.append(chr(key))

            #Append the current characters to a string to be displayed on
screen
            text = empty_string.join(current_list)
            #Run the displaytext module to display the text on screen
            self.displaytext(screen, x, y, size, text)

    def displaytext(self, screen, x, y, size, text):
        #Assigning font properties to a variable
        font = pygame.font.SysFont("calibri", 16)
        text = font.render(text, 1, black)

        #Getting the coordinates of where the text should start - Middle of
the input box
        middlex = x + (size[0]/2)
        middlex -= ((text.get_width()+1)/2)
        middley = y + (size[1]/2)
        middley -= (((text.get_height()+1)*(3/4.))/2)

        #Display the text on screen
        pygame.draw.rect(screen, (200, 200, 200), ((x, y), size))
        screen.blit(text, (middlex, middley))
        pygame.display.update()
```

Dijkstra's Algorithm

Here is the fully annotated code for the Dijkstra's algorithm file used in my program. This section of code contains the mathematical procedures required to solve the grid/ graph when given the appropriate data.

```
#Importing necessary modules
from math import log10, floor

#Assigning RGB values to variables
black = (0, 0, 0)
grey = (125, 125, 125)
white = (255, 255, 255)
red = (255, 0, 0)
green = (67, 205, 128)
brown = (139, 69, 39)

#The initial function for the Grid function
def main(size_of_grid, grid_references):
    #Creating an empty list to hold properties for each cell
    weight_list =[]

    #Loop through all the cell
    for i in range(size_of_grid[0]):
        weight_list.append([])
        for j in range(size_of_grid[1]):
            #Retrieve the colour of the cell and get the weight
            colour_of_cell = grid_references[i][j][2]
            weight_of_cell = get_weight(colour_of_cell)
            #Append an list for each cell containing the weight of the
cell, then 3 empty strings to be replaced with permanent value, working
cumulative weight and the permanent cumulative weight respectively
            weight_list[i].append([weight_of_cell, "", "", ""])

            #Assign the start and destination cell reference to variables
            if grid_references[i][j][2] == green:
                start_position = [i, j]
            elif grid_references[i][j][2] == red:
                end_position = [i, j]
                #Assigning weight of destination cell to 1
                weight_list[i][j][0] = 1

    #Use the assign values function to get fill in the missing properties
(permanent value, working cumulative weight and the permanent cumulative
weight)
    updated_weight_list = assign_values(size_of_grid, weight_list,
start_position, end_position)

    #Use the updated weight list, with complete properties, to find the
shortest route
    finished_route = find_route(size_of_grid, updated_weight_list,
start_position, end_position)

    #Return the shortest route so it is assigned to a variable
    return finished_route

#Get the weight of each cell from it's colour
def get_weight(colour_of_cell):
    #Grey cells are 0 - They are impassable
    if colour_of_cell == grey:
```

```
        weight = 0

    #White/ passable terrain cells are 1, normal weight
    elif colour_of_cell == white:
        weight = 1

    #Brown/ slow terrain cells are 2, they take twice as long to pass
through
    elif colour_of_cell == brown:
        weight = 2

    #This forms part of the validation
    else:
        weight = ""
    return weight

#Assigning the missing properties to the weight list
def assign_values(size_of_grid, cells, start, end):
    #Assigning the properties of the start cell to 0
    cells[start[0]][start[1]] = [0, 0, 0, 0]
    #When calling the cells list it follows this formula:
    #cells[X Position/ Column][Y Position/ Row][0= weight || 1= permanent
value (count) || 2= working cumulative weight || 3= permanent cumulative
weight]

    #Assigning start as the current cell and some place holder variables
    current_cell = start
    finish = False

    #Once all the permanent weights have been filled in, this looks at the
remaining weights and checks which has the lowest working weight and then
assigns all the cells with the lowest working weight into this list
    smallest_values = []

    #Until all cells are filled in
    while finish != True:
        #The options are right, down, left and above the current cell,
these check if the cell is accessible
        option1, option2, option3, option4 = False, False, False, False

        #These are the the [column, row] positions of each option
        right = [current_cell[0] +1, current_cell[1]]
        down = [current_cell[0], current_cell[1] +1]
        left = [current_cell[0] -1, current_cell[1]]
        up = [current_cell[0], current_cell[1] -1]

        #Checks to see if each cell is valid or not
        #If the option is not outside the grid, the cell hasn't already
been assigned a PERMANENT value and the cell isn't impassable terrain
        if right[0] < size_of_grid[0] and cells[right[0]][right[1]][0] != 0
and cells[right[0]][right[1]][1] == "":
            #Then that option is True - The cell can be accessed
            option1 = True
        #This continues for all the options
        if down[1] < size_of_grid[1] and cells[down[0]][down[1]][0] != 0
and cells[down[0]][down[1]][1] == "":
            option2 = True
        if left[0] >= 0 and cells[left[0]][left[1]][0] != 0 and
cells[left[0]][left[1]][1] == "":
            option3 = True
```

```
        if up[1] >= 0 and cells[up[0]][up[1]][0] != 0 and
cells[up[0]][up[1]][1] == "":
            option4 = True

        #Here I assign each of the options, if they are valid, certain
properties:
        #If the cell is valid
        if option1 == True:
            #If the working weight (the current permanent weight of the
cell, plus the distance to the next cell, is less than working weight of
the next cell
            if cells[current_cell[0]][current_cell[1]][3] +
cells[right[0]][right[1]][0] < cells[right[0]][right[1]][2]:
                #Replace the working weight with the new, lower working
weight
                cells[right[0]][right[1]][2] = cells[right[0]][right[1]][0]
+ cells[current_cell[0]][current_cell[1]][3]
                #Also the permanent value of that cell is the permanent
value of the current cell plus one - This shows which cell is the next cell
from it
                cells[right[0]][right[1]][1] =
cells[current_cell[0]][current_cell[1]][1] + 1
        #This continues for each available option
        if option2 == True:
            if cells[current_cell[0]][current_cell[1]][3] +
cells[down[0]][down[1]][0] < cells[down[0]][down[1]][2]:
                cells[down[0]][down[1]][2] = cells[down[0]][down[1]][0] +
cells[current_cell[0]][current_cell[1]][3]
                cells[down[0]][down[1]][1] =
cells[current_cell[0]][current_cell[1]][1] + 1
        if option3 == True:
            if cells[current_cell[0]][current_cell[1]][3] +
cells[left[0]][left[1]][0] < cells[left[0]][left[1]][2]:
                cells[left[0]][left[1]][2] = cells[left[0]][left[1]][0] +
cells[current_cell[0]][current_cell[1]][3]
                cells[left[0]][left[1]][1] =
cells[current_cell[0]][current_cell[1]][1] + 1
        if option4 == True:
            if cells[current_cell[0]][current_cell[1]][3] +
cells[up[0]][up[1]][0] < cells[up[0]][up[1]][2]:
                cells[up[0]][up[1]][2] = cells[up[0]][up[1]][0] +
cells[current_cell[0]][current_cell[1]][3]
                cells[up[0]][up[1]][1] =
cells[current_cell[0]][current_cell[1]][1] + 1

        #If there are no items in the smallest_values list - All the
permanent weights have been filled in
        if len(smallest_values) == 0:
            #Assigns infinity to a variable to check which is the lowest
number
            smallest_working_value = float('inf')
            #Loop through all the cells in the grid
            for i in range(size_of_grid[0]):
                for j in range(size_of_grid[1]):
                    #If the weight of a cell is smaller than the
smallest_working_value, it changes the smallest_working_value to the lowest
weight - This finds the lowest working weight out of any of the cells which
don't have a permanent weight
                    if cells[i][j][2] < smallest_working_value and
cells[i][j][3] == "":
                        smallest_working_value = cells[i][j][2]
```

```
            #Loops through all the cells and assigns those with the
smallest working weight to a list
            for i in range(size_of_grid[0]):
                for j in range(size_of_grid[1]):
                    if cells[i][j][2] == smallest_working_value:
                        smallest_values.append([i, j])

        #If there are no more smallest values and none of the options are
available function ends - This is either when we reach the destination cell
or there is no route to the destination cell and it hits a dead end
        if option1 == False and option2 == False and option3 == False and
option4 == False and len(smallest_values) == 0:
            finish = True

        #Whilst finish isn't True - there are still cells available
        if finish != True:
            #Assigns the current cell to the next value in the list of
smallest values
            current_cell = smallest_values[0]
            #Removes the current cell from the list - Since we are going to
check this now it is no longer going to have no permanent weight
            smallest_values.remove(current_cell)
            #The permanent weight of this cell is the smallest working
value
            cells[current_cell[0]][current_cell[1]][3] =
smallest_working_value

    #Return the cells with all properties assigned
    return cells

#Using the completed weight list with all properties assigned, find the
route
def find_route(size_of_grid, cells, start_position, destination_position):
    #Creating place holder variables
    finished_route = []
    #The first cell in the final route must be the destination cell
(working from destination to start)
    finished_route.append(destination_position)
    finish = False
    count = 0

    while finish != True:
        #Current cell is the next cell in the final route
        current_cell = finished_route[count]
        count += 1

        #Assigns cells adjacent to the current cell to variables
        option1, option2, option3, option4 = False, False, False, False
        right = [current_cell[0] +1, current_cell[1]]
        down = [current_cell[0], current_cell[1] +1]
        left = [current_cell[0] -1, current_cell[1]]
        up = [current_cell[0], current_cell[1] -1]

        #Here we check not only that the option is within the grid and
valid, but also that the permanent value of the current cell subtract the
permanent value of the cell we are looking at is 1 - This checks that it
was directly connected to it as part of the route
        if right[0] < size_of_grid[0] and cells[right[0]][right[1]][1] !=
"" and cells[current_cell[0]][current_cell[1]][1] -
cells[right[0]][right[1]][1] == 1:
```

```
            #The second part of this check, checks that the permanent
weight of the current cell subtract the permanent weight of the cell we are
looking at, is the distance between the two cells - Part of Dijkstra's
algorithm
            if cells[current_cell[0]][current_cell[1]][3] -
cells[right[0]][right[1]][3] == cells[current_cell[0]][current_cell[1]][0]:
                option1 = True

        if down[1] < size_of_grid[1] and cells[down[0]][down[1]][1] != ""
and cells[current_cell[0]][current_cell[1]][1] - cells[down[0]][down[1]][1]
== 1:
            if cells[current_cell[0]][current_cell[1]][3] -
cells[down[0]][down[1]][3] == cells[current_cell[0]][current_cell[1]][0]:
                option2 = True

        if left[0] >= 0 and cells[left[0]][left[1]][1] != "" and
cells[current_cell[0]][current_cell[1]][1] - cells[left[0]][left[1]][1] ==
1:
            if cells[current_cell[0]][current_cell[1]][3] -
cells[left[0]][left[1]][3] == cells[current_cell[0]][current_cell[1]][0]:
                option3 = True

        if up[1] >= 0 and cells[up[0]][up[1]][1] != "" and
cells[current_cell[0]][current_cell[1]][1] - cells[up[0]][up[1]][1] == 1:
            if cells[current_cell[0]][current_cell[1]][3] -
cells[up[0]][up[1]][3] == cells[current_cell[0]][current_cell[1]][0]:
                option4 = True

        #Once we have done this for each of the options, only one of them
is valid, so we append that to the final route
        if option1 == True:
            finished_route.append(right)
        elif option2 == True:
            finished_route.append(down)
        elif option3 == True:
            finished_route.append(left)
        elif option4 == True:
            finished_route.append(up)

        #If they are all invalid, that means we have reached the starting
cell and the final route is complete
        if option1 == False and option2 == False and option3 == False and
option4 == False:
            finish = True

    #We reverse the final route, so that it goes from start to finish and
return it back
    finished_route.reverse()
    return finished_route

#This is the function that handles the Vertices function of our program
def function2(line_coordinates, start_cell, end_cell):
    #Creating place holder variables
    #Cells is a list of all the cells that are used in the lines - Not to
be mistaken with the line coordinates
    cells = []
    current_perm_value = 0
    finish = False
    valid = True
```

```
    #Multiply the weight of each line by 10 to avoid floating point
arithmetic errors
    for i in range(len(line_coordinates)):
        line_coordinates[i][4] = line_coordinates[i][4] * 10

    #Creating a list of all the cells used in the program, with place
holder properties
    for i in line_coordinates:
        #Here we use the formula cells[column][row][0= x co-ordinate || 1=
y co-ordinate || 2= permanent value || 3= permanent cumulative weight || 4=
working cumulative weight] for retrieving data
        if [i[0], i[1], "", "", ""] not in cells:
            cells.append([i[0], i[1], "", "", ""])
        if [i[2], i[3], "", "", ""] not in cells:
            cells.append([i[2], i[3], "", "", ""])

    #Loop through all the cells
    for i in range(len(cells)):
        #Assign the start cell with a working cumulative weight of 0
        if cells[i][:2] == start_cell:
            cells[i][4] = 0
        #Retrieve the coordinates of the end cell
        elif cells[i][:2] == end_cell:
            end_cell_pos = i

    while finish != True:
        finish = ""
        current_perm_value += 1

        #Finding which cell has the lowest working cumulative weight which
hasn't been assigned a permanent value
        x = float('inf')
        current_cell = []
        for i in cells:
            if i[2] == "" and i[3] == "" and i[4] < x:
                x = i[4]
                current_cell = i

        #If there are no cells which are valid
        if current_cell == []:
            #The loop ends, but valid is set to False since there is no
route from the start to the end cell
            finish = True
            valid = False

        #Otherwise if there are valid cells
        else:
            #The current cell is given a permanent value and permanent
cumulative weight
            position = cells.index(current_cell)
            cells[position][2] = current_perm_value
            cells[position][3] = x
            #Update the current cell with new properties
            current_cell = cells[position]

            #Amy lines branching off the current cell are appended to the
vertex options list
            vertex_options = []
            for i in line_coordinates:
                if i[:2] == current_cell[:2]:
                    vertex_options.append(i)
```

```
                elif i[2:4] == current_cell[:2]:
                    vertex_options.append(i)

            #Loops through vertex options
            for i in vertex_options:
                #Calculates the weight to that next cell by adding the
cell's permanent weight to the next
                working_weight = i[4]+current_cell[3]
                #This works out the cell which isn't the cell we're
currently looking at, out of the two in the line co-ordinates
                if i[:2] == current_cell[:2]:
                    working_cell = i[2:4]
                elif i[2:4] == current_cell[:2]:
                    working_cell = i[:2]

                #Finds the other location of the working cell in the cells
list
                for i in cells:
                    if i[:2] == working_cell:
                        working_cell_position = cells.index(i)

                #If the cell doesn't have a working weight or the current
working weight is less the working weight the next cell it has
                if cells[working_cell_position][4] == "" or working_weight
< cells[working_cell_position][4]:
                    #The working weight of the next cell is replaced by the
current working weight
                    cells[working_cell_position][4] = working_weight

            #If any cells do not have a permanent cumulative weight, then
it doesn't finish
            for i in range(len(cells)):
                for j in cells[i]:
                    if j == "":
                        finish = False

            #Otherwise the loop is exited - All cells have a permanent
cumulative weight ready to be solved
            if finish == "":
                finish = True

    #This is the second section of the function - Here we create some place
holder variables
    #Here we start with the end cell and work backwards
    current_cell = cells[end_cell_pos]
    final_route = []
    finish = False

    #If the initial route was valid - If it is invalid the final_route is
left as [] so nothing is drawn for the solution
    if valid != False:
        while finish != True:
            line_options = []

            #Looks through the line coordinates to find which lines involve
the current cell and appends any lines connect to the current cell to a
list
            for i in line_coordinates:
                if i[:2] == current_cell[:2]:
                    line_options.append(i)
                elif i[2:4] == current_cell[:2]:
```

```
                    line_options.append(i)

            #Loop through line options list
            for i in line_options:
                #Loop through all the cells
                for j in cells:
                    #Finds the other coordinate of the line (the one which
isn't the current cell) and finds the cell properties
                    if i[:2] != current_cell[:2] and i[:2] == j[:2]:
                        #If the current cell cell permanent weight,
subtract the weight of the next cell is equal to the weight of the line
                        if current_cell[3] - j[3] == i[4]:
                            #Then this cell must be part of the final route
- The current cell is now assigned to this cell and it repeats
                            final_route.append(i)
                            current_cell = j

                    #Same as before but if the first coordinate is the
current cell
                    if i[2:4] != current_cell[:2] and i[2:4] == j[:2]:
                        if current_cell[3] - j[3] == i[4]:
                            final_route.append(i)
                            current_cell = j

            #Once the current cell is the start cell the loop ends
            if current_cell[:2] == start_cell:
                finish = True

            #Otherwise it continues unless there are no more line options -
Then the final route is set to [] so nothing is drawn for the solution
            elif line_options == []:
                finish = True
                final_route = []

    #Divides the weight of all the values by 10 to avoid any errors when
running the function twice and the weight being multiplied again and rounds
them to two significant figures
    for i in range(len(line_coordinates)):
        weight = line_coordinates[i][4] / 10.
        weight = round(weight, 2-int(floor(log10(weight)))-1)
        line_coordinates[i][4] = weight

    #Return the final route
    return final_route
```

References

[1]: http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html - 12/03/14