Visualisation of the Pathfinding Process in C++

Harry Cooke
3rd Year Physics with Particle Physics and Cosmology
School of Physics and Astronomy
University of Birmingham
Birmingham, B15 2TT

Friday 19th January, 2018

Keywords: Object Oriented Programming, C++, Pathfinding

ABSTRACT
An object oriented program is created to find optimal paths between points on a two-dimensional square or hexagonal grid, of user-defined size, and visualise the process by which the paths are found. The program may generate random obstacles or they may be defined by the user. If multiple goals are given, a variant of the travelling salesman problem is solved to give the optimal route travelling through all of them.

CONTENTS

1 INTRODUCTION - PROBLEM OUTLINE

The aim of this project is to create a graphical program in C++, using the Qt user interface framework, to find paths between given points on a two-dimensional grid. The grid should be of a size determinable by the user. There must be a start point and at least one finish point, as well as obstacles in between. The positions and quantities of these could be randomised or determined by the user. A path (ideally the shortest possible) between these start and finish points must then be found. This path should be shown to the user in some manner, as well as the total distance it covers along the grid.

As an additional goal, a key aim for this implementation is providing a visualisation of how the algorithm searches for the optimal path; which, combined with optional different pathfinding methods, allows for a qualitative comparison between algorithms.

This report outlines the design aims of the project, the implementation methods used, and the final result produced.

2 DESIGN SPECIFICATION

2.1 Interface

The key philosophy behind the interface design is simplicity. The program will contain just a single window, consisting of a control bar along the top and a display pane below, where the grid will be drawn, as illustrated by Figure 1. The size of the window will be fixed during runtime, to avoid complications with controls and with the grid visualisation, but could be determined by screen resolution at launch, in order to maximise available space for the user.

The control bar would ideally be relatively full, but if window width is determined by resolution then this is not always feasible. In such a case where the window is too wide for the controls to fill the control bar there should be a separation in the middle, with some controls on the left and some controls on the right, for a more symmetric appearance.

The display pane itself should be minimalist, ideally using only different coloured areas to present the necessary information. A lack of text or numbers may lead to sacrifices, in terms of features to further enhance pathfinding mechanics, but should greatly increase scalability in the size of the grid used; inclusion of text limits the size of any grid cell to a reasonable font size whereas a coloured area can be seen at a much smaller size, especially when grouped together with other similarly coloured cells.

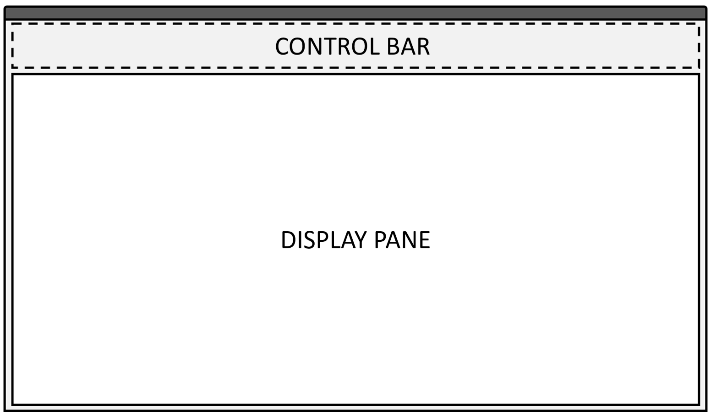

Figure 1. Basic layout of the program window. The control bar will contain all buttons, checkboxes, sliders, spin boxes, and drop down menus necessary to give the user control over key parameters in the program.

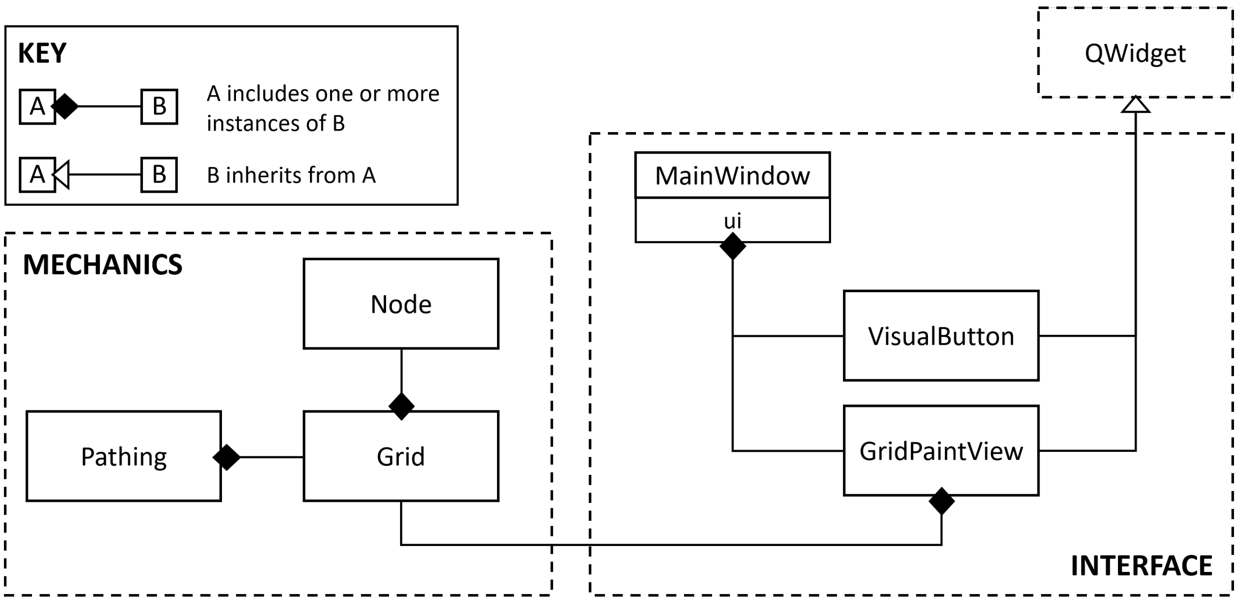

Figure 2. Structural diagram of interactions between classes in a simplified program layout. The interface consists of the window, MainWindow, and the its interface, ui, which contains all of the graphical components. VisualButton and GridPaintView are both custom classes derived from QWidget to perform specific tasks which are not available among default widgets. The display pane shown in Figure 1 would be an instance of the GridPaintView class and instances of VisualButton will exist within the control bar.

2.2 Code Structure

The underlying structure of the code should be made as simple as possible, like the interface, whilst also promoting adaptability to expansion and addition of features. In line with this, classes, properties, methods, and variables should be made general wherever possible, and not in reference to a specific number, state, or algorithm. For instance, a grid might typically consist of squares, but it could also consist of hexagons, triangles, or various combinations of shapes; therefore if a grid class is used it should only define properties necessary for any tessellating grid, anything specific to the mechanics of a square grid should belong to a derived class of this grid class.

To achieve this generality the grid class will consist of objects belonging to a node class. These nodes will have a defined type, either an obstacle, blank space, start, or goal, which will define how they act as well as how they appear on screen. The nodes will also have a vector of pointers to other nodes which are to be considered as neighbouring nodes by the pathfinding algorithm. This allows the neighbours of each node to be defined by the grid to suit its geometry, the pathfinding algorithm can then access the neighbours and run without particular concern for what type of grid it is running on; this is possible due to the polymorphic properties of object oriented programming.

Another key feature of the code structure should be a clear separation between the mechanical components, that handle the state of the grid and the pathfinding process, and the components of the interface, which handle the user input and output. This helps to mitigate the complexity of the program, effectively reducing it to two separate pieces of code with few links between.

Figure 2 sets out a structure for the program. The separation of mechanics and interface can clearly be seen here with only one link between, allowing the display pane to access the data in the grid in order to ‘paint’ it on the screen. The connections shown here are only the major links between classes, i.e. where an instance might be included as a data member. There are other methods of communication not represented here, such as the signals and slots provided by the Qt framework, which will doubtlessly be needed to connect the two sets of objects.

This diagram does not detail specific subclasses which might be needed to promote scalability, such as a square grid type or a specific pathfinding algorithm, as the number of these implemented really depends only on time. Having multiple pathfinding algorithms should be the first priority among these, as the visualisation will help to highlight the key differences in these algorithms.

3 CODE DESIGN REVIEW

The first iteration of the design phase of the project looked significantly different to that presented above. Initially the grid was going to be a strictly defined two dimensional vector of booleans, with true representing an obstacle and false representing a blank cell. With this setup, for an element of indices (i, j) in the 2D vector, the pathfinding algorithm would take the elements $(i+1, j)$, $(i-1, j)$, $(i, j+1)$, and $(i, j-1)$ as its neighbours, unless defined differently based on a conditional statement.

It was also planned for the start and goal to be instances of their own unique classes. The goal class was going to run the pathfinding process, which was not a logical arrangement as the goal is really just a point on the grid. The start, originally called the seeker, was going to move along the grid, finding its optimal path from the information provided by the goal but using some sort of line of sight feature which limits how far it can travel at once.

This arrangement was trying to be more of a simulation for a person optimally exploring a maze. In the code design review meeting, Prof. Lazzeroni and Dr. Slater went over this initial plan with me. Some of the key flaws in the logic and structure were highlighted, in particular the grid being defined in such a restrictive manner, the pathfinding process being controlled by the goal, and even the start and goal to have their own classes when they can just be properties of the grid.

The definition of the grid seen in the code now is a direct result of this meeting, as well as the Pathing class which is now independent of the goal node. The Node class allows any grid cell to be blank, an obstacle, the start node, or an end node; as well as storing the neighbours defined by the grid. Combined, this removes the need for either the start or goal to have an independent class.

4 THEORY

Pathfinding is a manifestation of the shortest-path problem in graph theory. This is an area which has been thoroughly explored and, as such, there are many different methods available to find the shortest path between two points; a selection of these methods are investigated here. Each of the methods has various advantages and disadvantages in terms of simplicity, efficiency, and speed.

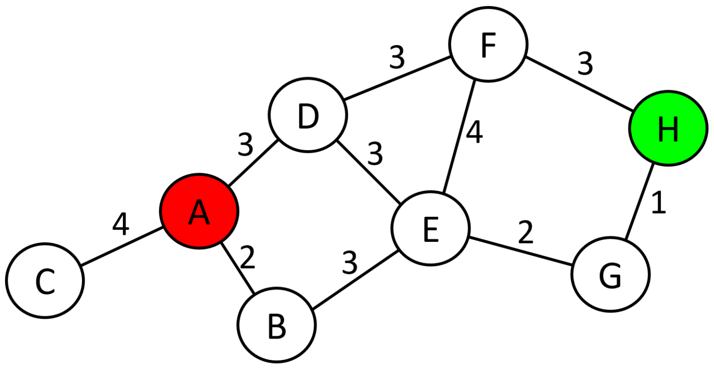

Figure 3. A weighted connected graph with highlighted start and end nodes, A and H respectively.

Each of these pathfinding algorithms iterates over different nodes and calculates the distance from the start node to those neighbouring the current node. The start node will be assigned a distance zero and then all of its neighbours assigned a distance one and pushed onto a queue. The key difference between the different algorithms is the order in which they choose to evaluate the nodes in the queue.

4.1 Breadth-First Search

One of the simplest methods for finding the shortest path is known as a breadth-first search [1]. In this algorithm the queue is treated as any standard queue, a first in first out (FIFO) data structure. This means that nodes are evaluated in order based on the number of edges they are from the start, irrespective of the weighting of those edges.

For the graph shown in Figure 3, the three nodes, B, C, and D, neighbouring the start, A, would be evaluated in any order (implementation dependent, based on which ones are discovered first). The set of their neighbouring nodes, in this case just E and F, would then be evaluated, followed by H and G. The algorithm stops as soon as it finds the endpoint, H. In this case that may happen before the edge between G and H is discovered, and so the algorithm could give A-D-F-H as its chosen path. In reality the path A-B-E-G-H is one unit shorter.

The main drawback of a breadth-first search is therefore that it will not always find the shortest path available. It is also particularly slow at finding a path compared to other algorithms. However, the example given is for a weighted graph; if the graph is unweighted then a breadth-first search will always find the optimal path. The primary advantage of this algorithm is that, due to the simple treatment of the queue, it is more straightforward to implement than the others discussed here.

4.2 Dijkstra's Algorithm

Dijkstra's algorithm, named after computer scientist E.W Dijkstra [2], treats the queue of nodes as a priority queue, so that the nodes which are closest to the start node are evaluated first. This means that a node is only reached by the algorithm when every node closer to the start has already been reached; by extension Dijkstra's algorithm will always find the shortest path between two nodes on any graph, weighted or unweighted. If the graph is unweighted then neighbouring nodes will all be the same distance away, in this case Dijkstra's algorithm is equivalent to a breadth-first search.

For the graph shown in Figure 3, the nodes will be evaluated in order A, B, D, C, E, F, G, H; it will return the correct shortest path of A-B-E-G-H.

Thus Dijkstra's algorithm is more effective than a breadth-first search in that it will always find the shortest path on a weighted graph. It will find a path in a comparable time to a breadth-first search also, although it will require more operations as it has to compare priorities of nodes in the queue. The priority queue means that it is more difficult to implement than a breadth-first search however and, in the case of an unweighted graph where the effect of the algorithms is the same, Dijkstra's may be slower to find the same path.

4.3 Greedy Best-First Search

Like Dijkstra's algorithm, the greedy best-first search also uses a priority queue to select the next node to be evaluated. In this case, the criteria evaluated is the distance to the goal instead of to the start, with the queued node closest to the goal being evaluated next [3]. As the goal has not been mapped at this point, the distance to the goal is not precise, but is instead an estimate known as a heuristic. If this heuristic was entirely accurate then the path found would always be the fastest, but without an ideal heuristic (which would render the pathfinding process redundant) the algorithm can return a sub-optimal path.

For the example in Figure 3, if the heuristic were to be based on the physical positioning of the nodes within the image, node D would be evaluated first. This would then add E and F to the queue, and F would likely be found closer to the goal. The path returned would therefore be A-D-F-H, which is one unit longer than necessary.

The greedy best-first search works well if the optional paths are very simple and the heuristic is a good approximation for the actual distance. It will also find a path faster than all the other algorithms discussed in almost every instance. This is offset by the path often being longer than other possible paths (and to find out if the path found is optimal another, slower, algorithm would have to be run anyway). In terms of implementation the greedy best-first search is marginally more complicated than Dijkstra's algorithm due to the required heuristic.

4.4 A* Pathfinder

It is possible find a balance between Dijkstra's algorithm, which explores all nodes within the radius of the distance to the goal, and the greedy best-first search, which does not explore enough nodes to find the optimal path. This balance is the A* pathfinder algorithm [3]. The queue for A* uses priorities based on minimising the sum of the distances to the start node and the end node. The node for which this sum is smallest will be evaluated next.

If run on the weighted graph in Figure 3, A* would evaluate nodes B and D first as they are close to the start and in the direction of the goal. Node E would then most likely be evaluated next, then C, then G, then H potentially, depending on the exact heuristic used. From this it would find the correct shortest path of A-B-E-G-H, without having examined every node (F is not examined).

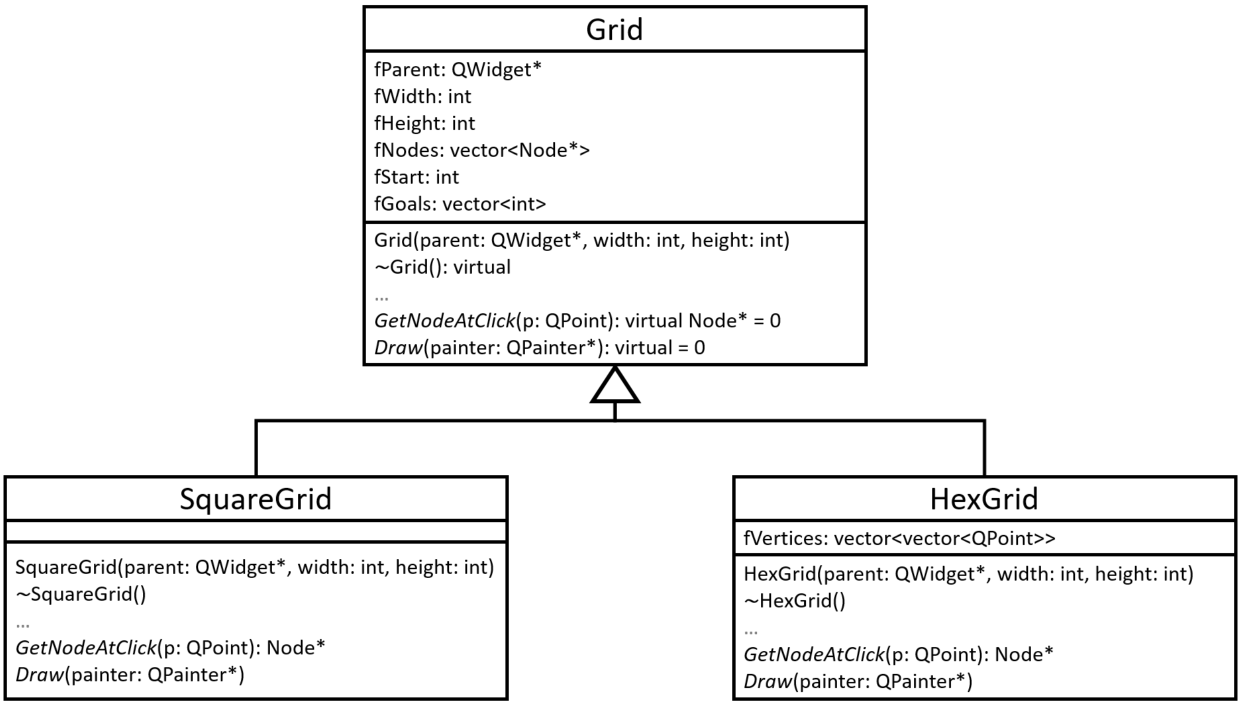
Grid
fParent: QWidget*
fWidth: int
fHeight: int
fNodes: vector<Node*>
fStart: int
fGoals: vector<int>
Grid(parent: QWidget*, width: int, height: int)
~Grid(): virtual
...
GetNodeAtClick(p: QPoint): virtual Node* = 0
Draw(painter: QPainter*): virtual = 0
SquareGrid
SquareGrid(parent: QWidget*, width: int, height: int)
~SquareGrid()
...
GetNodeAtClick(p: QPoint): Node*
Draw(painter: QPainter*)
HexGrid
fVertices: vector<vector<QPoint>>
HexGrid(parent: QWidget*, width: int, height: int)
~HexGrid()
...
GetNodeAtClick(p: QPoint): Node*
Draw(painter: QPainter*)

Figure 4. A simplified UML class diagram showing the classes derived from Grid. Both SquareGrid and HexGrid override the virtual functions ‘GetNodeAtClick’ and ‘Draw’ from the base class.

The A* algorithm is optimised such that it will always find the shortest available path between two nodes on a graph, weighted or not, as long as it uses an optimistic heuristic; one where the distance to the goal is never overestimated. It is faster than the other optimal path-finding algorithms and will examine fewer nodes to find the path in almost every situation. This comes in tandem with a slightly more difficult implementation than the other algorithms, as it requires both a heuristic and distance from start to order the queue.

4.5 The Travelling Salesman Problem

If a grid has multiple goals through which the path needs to go, finding the optimal path which visits all of them is an entirely separate problem. The first step to this is to find the distance between each of the nodes that need to be travelled through; this can be done with multiple instances of one of the pathfinding algorithms discussed above. Once the distances between each of the paths has been established the problem is reduced to a weighted complete graph, and the optimal path must travel through every node on this graph.

This becomes a variant of the travelling salesman problem (TSP). The travelling salesman problem deals with weighted complete graphs and how to find the Hamiltonian cycle with the minimum weight [4]. The TSP is an NP-hard problem, meaning it is not known how to reduce it, in general, to a form that can be solved in polynomial time. There are solutions involving heuristics or iterative improvements that can be used to solve for near-optimal paths, but the only way to guarantee an optimal solution is to try every possible permutation. Given that there are $n!$ permutations of n nodes, this brute force solution to the TSP runs in time $\mathcal{O}(n!)$.

The problem in question, for multiple goals on the grid, varies slightly from the travelling salesman problem in that the required path is not a cycle and there is a fixed starting node. The solution is no more simple however, and still requires the brute force approach for the optimal route.

5 IMPLEMENTATION

The design detailed in section 2 has been implemented to a great degree of success; all of the essential features are included and operational. The structure of the code matches that proposed in figure 2, with the only difference being added subclasses to the Grid and Pathing classes.

5.1 Naming Standards

Various standards are used throughout the program for naming code elements; this section briefly outlines these standards. Classes and their methods use Pascal case (`SomeClass`), which is commonly used in object oriented programming. This also helps to distinguish between self-written methods and methods included by Qt, or redefinitions of these, which use camel case (`someMethod`). Properties of classes use similar casing but preceded with an ‘f’, for field, in Hungarian notation (`fSomeProperty`). Other variables and parameters use standard camel case, with rare exceptions for when a variable name is mathematical and uses snake case (`mathematical_variable`). Any macros used use the standard macro case (`SOME_MACRO`).

5.2 Grids

The Grid class has been implemented to the specification set out in section 2.2. This gives the flexibility for two different subclasses, SquareGrid and HexGrid, to be used interchangeably. Figure 4 shows how these two are related to the original grid class. Grid itself is an abstract class as it has two pure virtual functions. SquareGrid and HexGrid each override these two functions to give independent functionality. This allows for code to create a generic grid, of type Grid, which is created as either a SquareGrid or a HexGrid. Any methods declared in the Grid class can then be applied to this generic grid and the result will be dependent on which kind of grid was created. This polymorphism enables user

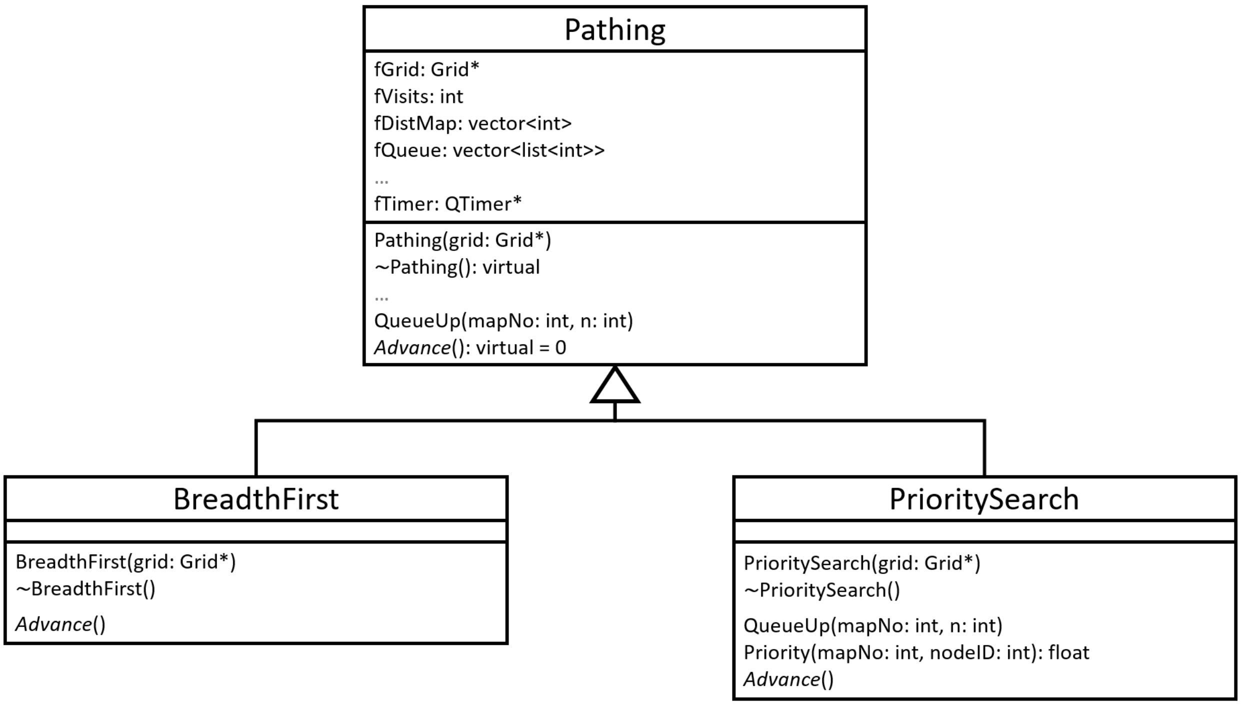

Figure 5. A simplified UML class diagram showing the classes derived from the Pathing class. Both BreadthFirst and PrioritySearch override the virtual function ‘Advance’. PrioritySearch also overrides the ‘QueueUp’ access method, defined in Pathing, so that it may implement a priority queue.

input of the type of grid without a large amount of duplicate code or conditional statements throughout.

The other key difference between the two derived grid classes is how the neighbouring nodes are defined, which occurs in the constructors of the classes. This is the key part of the mechanics to allow simple pathfinding on a generic grid. The overridden methods are for the different interactions with the interface. Both of these classes have rather complex Draw methods; this is to account for rounding errors when the cell sizes are cast to integers. The Draw method for HexGrid requires a two-dimensional vector of vertex points to be defined for this process; this vector is defined in the constructor instead of the Draw method itself to avoid unnecessary processing.

5.3 Pathfinding Algorithms

In a similar manner to the grid, the Pathing class has been implemented generally as an abstract class, with two derived classes defining different kinds of pathfinding algorithm. The Pathing class contains a timer which, when started, will continually call the Advance method. This is the method which is reimplemented by Pathing’s subclasses to modify the style of algorithm used. This timer and advance approach is used, in place of simply finding the path in one go, to enable the visualisation of the process. For both subclasses, the Advance function is written such that it will work with either one goal or many.

The subclass BreadthFirst uses the breadth-first search algorithm to shape its Advance method. On the other hand, PrioritySearch may take the form of A*, Dijkstra’s, or greedy best-first search, by modifying two weight properties defined in the Pathing class. The user may choose to apply either Dijkstra’s algorithm, an A* pathfinding algorithm, or a greedy best-first search to their grid. Due to the lack of node weights present in this implementation, Dijkstra’s algorithm is actually run by the BreadthFirst class instead of PrioritySearch meaning that it is technically a breadth-first search instead. However, as discussed in section 4, on such an unweighted grid the breadth-first search is simply a more efficient

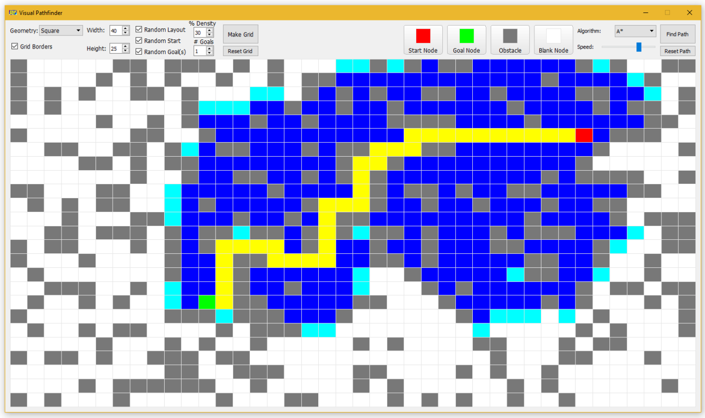
Visual Pathfinder
Geometry: Square
☑ Grid Borders
Width: 40
Height: 25
☑ Random Layout
☑ Random Start
☑ Random Goal(s)
% Density 30
Goals 1
Algorithm: A*

Figure 6. The implemented interface of the application. There is a control bar along the top, split down the middle for symmetry, and a display pane in the centre, using only colours to convey information; as specified in the original design.

Disjkstra’s algorithm. The layout of the pathfinding classes would easily allow for both Dijkstra’s algorithm and breadth-first search to be included as options if there was a significant difference between them.

5.4 Interface

The final interface design follows the specification very closely. The window size is determined by screen resolution, with a minimum width built in to ensure all the controls will fit. The controls on the right hand side are fixed to that edge, no matter what width the window ends up being.

The display pane has maintained a minimalist theme, only using colours to present the pathfinding information. This means that the grid size is very scalable. A grid size limit is imposed as 999 by 999, partially to avoid potential overflow issues and partially because it is a convenient limit for the spin box, and it is possible to run the pathfinding process on this grid and roughly follow what is happening.

In order to maintain the minimalist interface, when the pathfinding process is complete, a message box pops up to inform the user of the distance of the path found; or conversely to inform of the inability to find a path if it was not possible.

There are interactive buttons on the right hand side of the control bar used for editing the nodes on the grid. These buttons are instances of the VisualButton class, and the drawn shape on them will change with the geometry of the grid.

6 LIMITATIONS

All of the pathfinding algorithms available are implemented such that they will run if there is just one goal or if there are many goals. However, the algorithms which use heuristics currently only only focus on a single goal node. These algorithms are still capable of finding the distances between endpoint nodes but have been deactivated, if multiple goals are present, as the heuristics become meaningless if the node it is searching for is not the one the heuristic points it towards.

For a path with multiple goals, if one of the goals is not reachable, but all others are, the program returns a failed search. This could be improved such that it finds the optimal path through all reachable goals.

Currently, the number of goals that may be used in any one path is limited to 10. This is due to the $n!$ relationship between the number of goals and time; with 10 goals there is a noticeable lag of more than a second or so. If there were 15 goals this lag would become over 100 hours.

7 IMPROVEMENTS AND EXTENSIONS

Following from the existing limitations of the program, it would be interesting to see a heuristic algorithm acting on a grid with multiple goals. This could potentially focus on one goal at a time, then move the heuristic focus to a different goal once that one has been found. It would not have the same efficiency as A* manages on just a single goal however, and it could even be worse than Dijkstra's algorithm.

Another extension of a current feature could be the inclusion of a different algorithm used for the travelling salesman problem part of multiple goal pathfinding. This algorithm could possibly be used only if the number of goals was above 10, although it would not always find an optimal path.

Currently, the pathfinding process implemented here never evolves beyond an unweighted graph. It would be possible to add weighted nodes, with a larger cost to travel through them. The main challenge of this would be including it within the minimalist interface; to allow any depth of weights would really require numbers visible on the nodes. It could be possible to give just one different weighting, i.e. double the current weight, and assign weighted nodes a different colour. To attempt to include multiple weights a gradient of colours could be used; although this would be inexact and could make the process more difficult to understand.

It would be possible to depart entirely from grids,and allow the user to place their own nodes and define connections between nodes. The connections, or edges, could take user input for weights attached to them, although this would conflict with the minimalist interface again. Another option would be to have the weights of the edges defined by the on-screen distance between them. Implementation of this free-node setup would be feasible, and a lot of the functionality of the Grid class could be used to do it, but there would be a lot of things in the Grid class that it would not use. The Grid class would then have to be generalised to some sort of abstract node group class, moving things like the node storage (which is not dependent on a two-dimensional coordinate system) and the neighbours, but leaving the overloaded access methods which are coordinate dependent in the Grid. Deriving this node group class to a node network class would achieve this implementation.

If a node network was successfully implemented, it could be extended to solve some real world pathfinding problems. If the program allowed an image to be uploaded, and set as a background for the display pane, then paths could be traced over a satellite image with nodes and edges. This could allow the program to give fastest routes between real locations; potentially, on small scales, in instances where something like Google Maps might not be able to.

8 CONCLUSION

The aim of this project, to visualise the pathfinding process, has certainly been achieved. A program has been made which will randomly generate, or allow the user to define, a two-dimensional grid, in either squares or hexagons. A path can then be found between a given set of start and goal points, with an optimal path travelling through as many as 10 different goals being found if requested.

The specification laid out has been followed closely, and the program achieves everything it needed to and more. The limitations among the current features are highlighted and a substantial set of extensions is proposed.

REFERENCES

[1]Moore E.F. (1959). 'The shortest path through a maze', *Proceedings of the International Symposium on the Theory of Switching, Harvard University Press*, pp. 285292.

[2]Dijkstra E.W. (1959). 'A note on two problems in connection with graphs', *Numerische Mathematik* (1), pp. 269271

[3]Zhang H. (2014). 'Informed Search Methods', *University of Iowa* [online]. Available from `http://homepage.divms.uiowa.edu/~hzhang/c145/notes/chap3b.pdf`. [Published 23/09/2014.]

[4]Casquilgo M.A.S (2012). 'Travelling Salesman Problem', *Technical University of Lisbon* [online]. Available from `http://web.tecnico.ulisboa.pt/~mcasquilho/CD_Casquilho/TSP_Prototype.pdf`. [Published 16/06/2012.]