

Simulation of a Fire Engine Response

Anna Wilde

14820868

Chapter	Page
Introduction, 1	2
Approach, 2	3
Implementation, 3	3
Literature Review, 4	7
Testing, 5	7
Conclusion, 6	8
Appendix, 7	9
References, 8	10

1. Introduction

1.1 Functionality of the System

The system demonstrates a simulation of a *fire engine* extinguishing multiple instances of *fire* across a city. The city is modelled as a two-dimensional network of nodes in a uniform 6x6 grid. The fire engine continuously attempts to find the shortest path to the fire using the A* path finding algorithm. Once the path has been found, the fire engine traverses to the fire to *put the fire out*, the system then randomly generates a new position for the fire and the algorithm starts again, finding the path from the fire engine's current position.

1.2 Changes from the Original Proposal

The original proposal detailed a more advanced crime-detection simulation. Implementing all of the features of this system would have been an arduous challenge, hence why I scaled down the project and simplified some of the processes: the new system only has to manipulate one moving agent instead of the proposed two, and the agent has fewer operations on the nodes. In the new system, the nodes do not have interactive properties and exist only to represent the travel space. Another alteration is that I wanted the new system to also show the process of the search algorithm; to visualise how the agent is finding the shortest path. I thought that this would be a challenging enough addition to compensate for the minimisation of agent functionality.

1.3 Inputs and Outputs

The program randomly generates the positions of the fire and the actions of the fire engine are automatic, so there are no substantial user inputs to the system. The program outputs a visualisation of the A* algorithm calculating the path in real time, and shows the movement of the fire engine along the path.

1.4 Real World Problem

This simulation offers insight into how real-time path finding systems on fire engines could optimise the speed at which they prevent fire damage to the city. My single agent system demonstrates a linear view of a single engine navigating to fires in the fastest route from its dynamic position. Future implementations using multiple agents would demonstrate how this system could improve efficiency by navigating individual fire engines to the closest crisis.

2. Approach

2.0 Planning

I chose to use the NetworkX library as it seemed ideal for my system; I could set up the civilians as a traversable graph through NetworkX nodes and weighted edges, and NetworkX nodes have the ability to store any type of extra information in a Python dictionary-like syntax, which is advantageous in my state-oriented system.

Whilst familiarising myself with the NetworkX documentation, I discovered an example of a NetworkX weighted graph using the Matplotlib library to visualise the NetworkX graph (NetworkX 2017). Under further investigation, I discovered that Matplotlib also contains a built-in animation module (Matplotlib 2002), so it was clear that the library would be compatible with the data I was manipulating in the program and would be capable of displaying it in the way I had intended.

2.1 Path-Finding Algorithm

There are many path-finding algorithms which I could have used, but I chose A* as it is the best algorithm to apply to scenarios where finding the shortest path is not the only goal; in my simulation the focus is on getting to the destination as fast as possible, which requires an algorithm which finds the shortest path in a timely manner. A* completes this process faster than other algorithms such as Dijkstra's algorithm as it calculates which of the available edges is the most likely to be in the shortest path. This calculation is enabled through a heuristic which estimates the cost from the current node to the goal (Patel, Introduction To A* 2014).

3. Implementation

3.1 Initial Setup

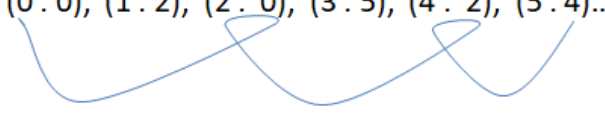
After installing the libraries, I set up a primitive graph of 5 connected NetworkX nodes and displayed them^{APP[1]} using Matplotlib, by adapting the example I had found on the NetworkX documentation (NetworkX 2017).

3.2 Building the Algorithm

The algorithm begins with a given start node and considers each edge connected to this node in turn. It calculates the distance to the connected node, and if this edge found is the shortest distance known from the start node then the edge relationship is added to a dictionary of *closed nodes*. The algorithm then selects another node from the list of possible known nodes and restarts this loop. As the dictionary is updated every time a shorter *distance to node x* is found, the final version contains a masterlist of the fastest way to get to any node in the network. To retrieve the shortest path from the node to goal, the function loops through the key:value pairs of the dictionary, working backwards from the goal.

So for example:

{ (0 : 0), (1 : 2), (2 : 0), (3 : 5), (4 : 2), (5 : 4)..... }



In this example, this is the resultant dictionary for *closed_nodes*, where the start was 0 and the goal was 5. The retrieval method would use *end* as the first dictionary key, and see that the shortest path edge was the link through 4, so the relationship (5, 4) would be added to the shortest path. Using 4 as the next key, we see that the next edge is (4, 2), then (2, 0) to reach the goal. So the final path is (0-2-4-5).

My first version of the algorithm implementation was just this function, without the heuristic. This process alone did successfully find the shortest path, but only through considering every single node in the network; a brute-forced approach^{App[2]}.

To optimise the speed of the algorithm, the algorithm needed a heuristic, which would reduce processing time by prioritising the edges which were likely to be relevant to the final path. This was quite an overhaul for the system as until that time, I had not assigned coordinates to the nodes, and was instead working purely from the connectivity of the network. I added three new attributes to all the nodes in the network: a tuple (x,y) of the coordinates, the x coordinate, and the y coordinate separately. The NetworkX documentation (NetworkX 2017) helped me to achieve this. The tuple was used to draw the nodes in their fixed positions. The separate x and y coordinates were for the algorithm to access, to calculate the heuristic distance estimate.

I used the Diagonal distance heuristic, as my graph is a square grid which allows eight directions of movement (Patel, Heuristics 2011). This heuristic estimates the number of steps required in the x and y plane, and then subtracts the distance you would save by traversing diagonally. Once calculated, this estimate is combined with the distance to get from the start to the current node, and is saved as the value in another dictionary,

priorityList. When the algorithm progressed to selecting the next edge to consider, it selects the key to the lowest value (distance + heuristic estimate). By adding a line to break the loop if the goal node is reached, the algorithm successfully found the shortest path to the goal, by only considering edges which were in the general vector from start to goal.

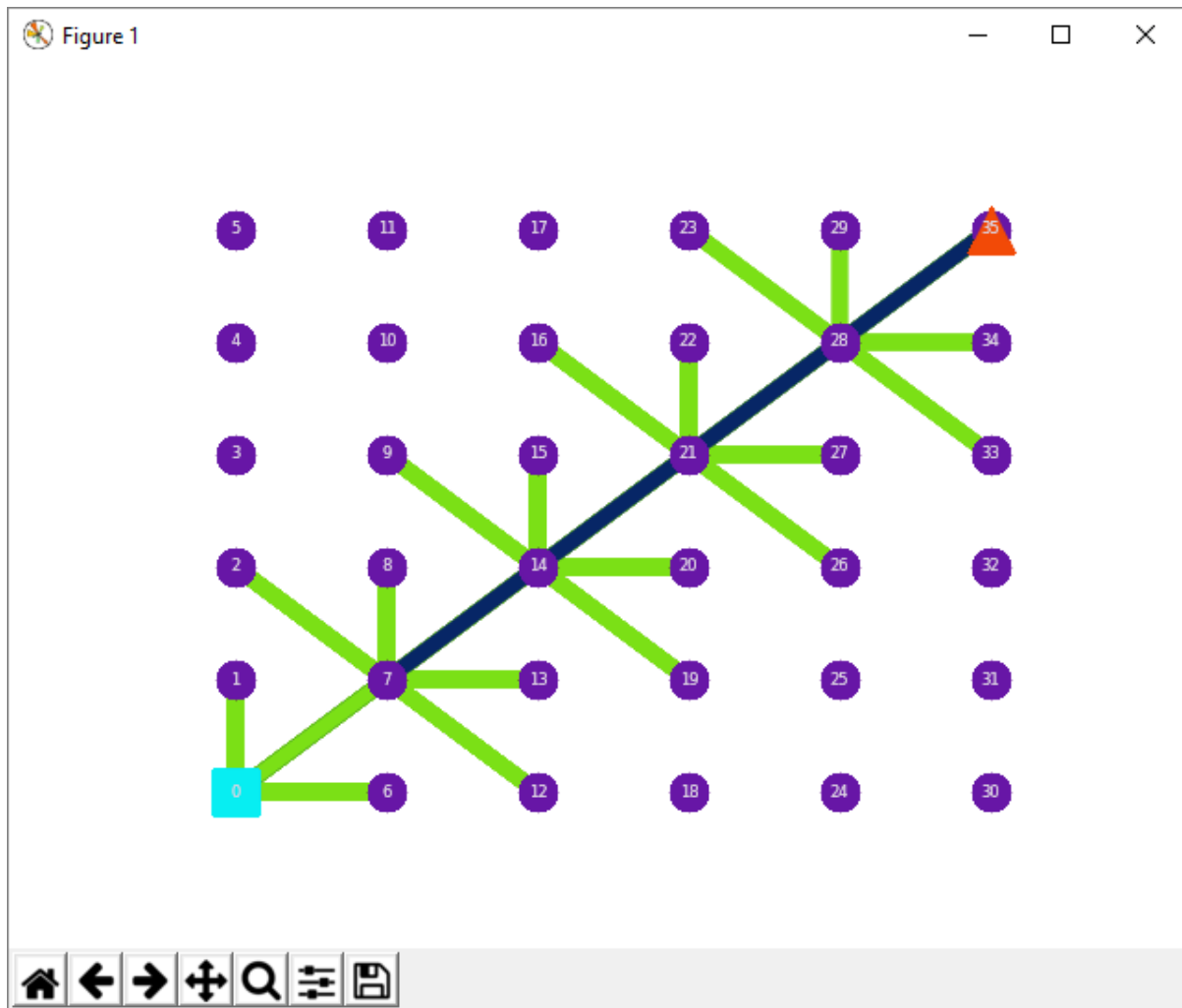
3.3 Matplotlib Display

With a successful algorithm, the last programming task was to modify the Matplotlib output to display the algorithm at each step, and show agent movement. I achieved this by drawing the setup of the graph before the algorithm was called, and then updating during every loop of the algorithm. I considered using the library animation module, but this module requires a definition of a consistent graph change, for example a sine graph. As my animation would be unpredictable, I decided to define an original animation method. The method draws the major graph components each time it is called, based on the parameters it has been passed. During the algorithm, two lists are involved: *eVisited* which is a list of the edges which the algorithm has considered, and *ePath*, which is the list of the final shortest path. By calling the animation method during the consideration loop, it would redraw the whole list every time, but the user would only notice the addition of the latest item in the list to the graph. The same process occurs for the final path found; the animation method is called as the edges are retrieved from the *closed_nodes* dictionary.

The last step was to add the visualisation of the agents and modify the program to run on a loop. I struggled to find a way to separate nodes for the *fire* and *fire engine* as I define the positions of the nodes automatically for all nodes, and the agents needed to be able to move dynamically. NetworkX allows the nodes to be represented in a variety of shapes, and any hexadecimal colour, so I decided to represent the fire and fire engine using the existing nodes. The animation function would take two extra parameters of node references, for the fire and fire engine, and display those nodes as different colours and shapes to the rest of the nodes. When the function is called after the position of the agents change, their previous position would be redrawn as a standard node, showing the user a movement of the fire engine.

I modified the program to run the main method in a loop, where at the end of the loop, the current position of the *fire engine* is used as the 'start' node for the next loop, and a new location for the *fire* is randomly generated, this is used as the 'goal' node.

The Matplotlib display shows the network of nodes, the *fire engine* as the blue square, and the orange triangle as the *fire*. The edges considered by the algorithm are shown in green, and the final shortest path found is represented in blue.



3.4 Challenges and Resources:

- ✓ I struggled to assign positions to the nodes, as I was drawing the nodes in a default *spectral layout*. I asked for help on StackOverflow and was given a suggestion on how to assign the positions generically (StackOverflow 2017).
- ✓ My program produces a Matplotlib Depreciation Warning. I investigated whether this was a concerning result, as the program seemed to be functioning fully despite the error. I found that it is not considered a major problem (StackOverflow 2014).
- ✓ I found that I could use standard characters to define the node shape (so[^]>v<dph8) (StackOverflow 2015).
- ✓ I needed to utilise the shortest path found to show *fire engine* movement, but my program saves the list in the reverse order (Goal to Start). After reviewing the Python documentation, I found that I could reverse the list by using `list.pop(i)`. If 'i' is not specified, then `list.pop()` removes the last item from the list (Python 2017).

- ✓ When the main program ran repeatedly, I needed to find a way to clear the graph between loops without closing the graph window. I found that I could use `clf()` to clear the figure (StackOverflow 2017).
- ✓ I used a web design website to select the hex colours to use in my graph (WebPage FX 2017).

4. Literature Review

Amitha Arun wrote a thesis on pathfinding algorithms in multi-agent systems. She outlined some of the variants of the A* algorithm, such as Theta* algorithm. The shortest path returned by A* is blocky and unrealistic for agent movement. Theta* can return an improved path by not constraining the path to graph edges, but allow the parent edges to be from any angle (Arun 2014). Theta* was introduced for the purpose of creating true short paths which are also realistic, as my program is a simulation of real life events, it would be an improvement to implement Theta* as this would show a more life-like movement of the fire engine to the goal. Arun designed a system which implements a multi-agent system similar to that which I originally proposed; this is achieved through an object-oriented style program which contains agents as objects, and path-finding algorithms as separate methods. This is a more suitable approach than my linear design, as my design is only capable of path-finding for one individual agent, and cannot process movement until after the path has been found.

5. Testing

The program runs on a loop of 30 iterations, where the fire is randomly relocated on each iteration. As I have executed the program multiple times and have not received any path-finding errors, it is safe to assume that the algorithm works for all possible start and goal positions.

Further testing will occur when I change the program to function for multiple agents; useful tests may include speed tests to ensure that each agent can process paths of the same distance at a similar speed. Another feature I may test is to implement multiple algorithms, such as A* and Theta*, and compare the efficiency and realism of the identified paths.

6. Conclusion

In my project proposal I outlined several measures for success. Not all of these are relevant to the finish program, as the project scope was altered during development. I did state that I will measure success with procedural outputs which show that the algorithms at work are successful (perhaps shown through a readable log of events); I have surpassed this measure of success as my program shows a visualisation of the algorithm's calculations in real-time, as a component of the animation. It may be directly seen that the algorithm is correctly considering the network's path based on the heuristic priorities, and does calculate the shortest path.

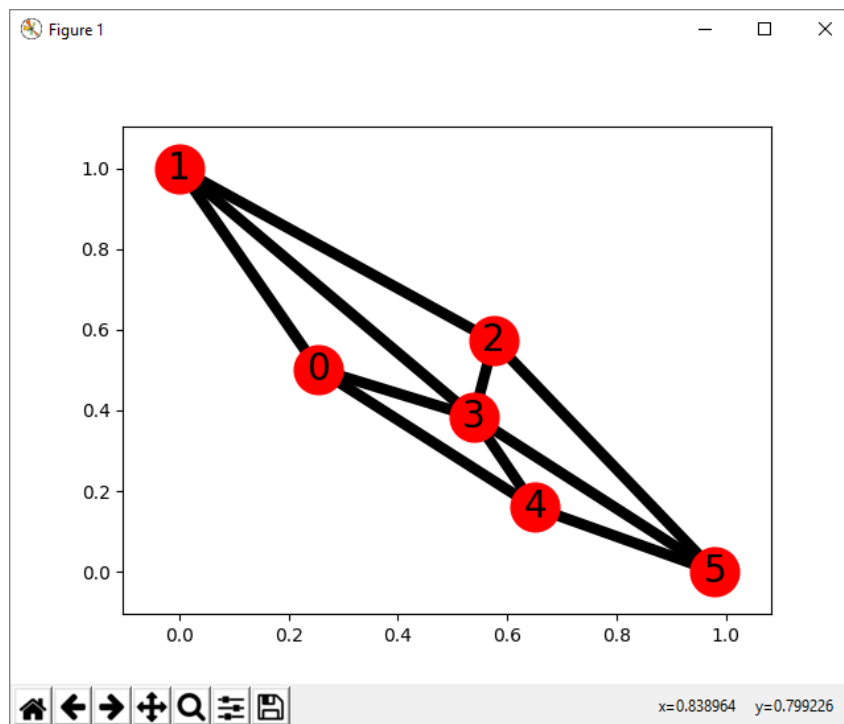
The next area for development is to modify the system to be multi-agent. This would require changing the architecture of the program to be non-linear; the algorithm process is completed in one step, but in a multi-agent system it may be necessary to enable the algorithm to return progress within every *frame* of the animation. This could be achieved by returning the list of open and closed nodes and visited edges, when the function is recalled this data would be passed back to the algorithm, and when the algorithm is initialised it would be passed empty lists only.

Another development area is to facilitate realistic movement: the current program allows the movement of the fire engine by traversing one edge per frame. As the diagonal edges are a greater cost than the planar edges, this is unrealistic. If the agent was to keep a consistent speed, it would only be able to travel a portion of the diagonal length during a frame. This development would require the separation of the agents from the nodes. This could potentially be achieved by separating the node network into two entities; static nodes and agent nodes. This level of separation can be achieved for edges using NetworkX ebunches (NetworkX 2015), so a similar process may be possible for nodes.

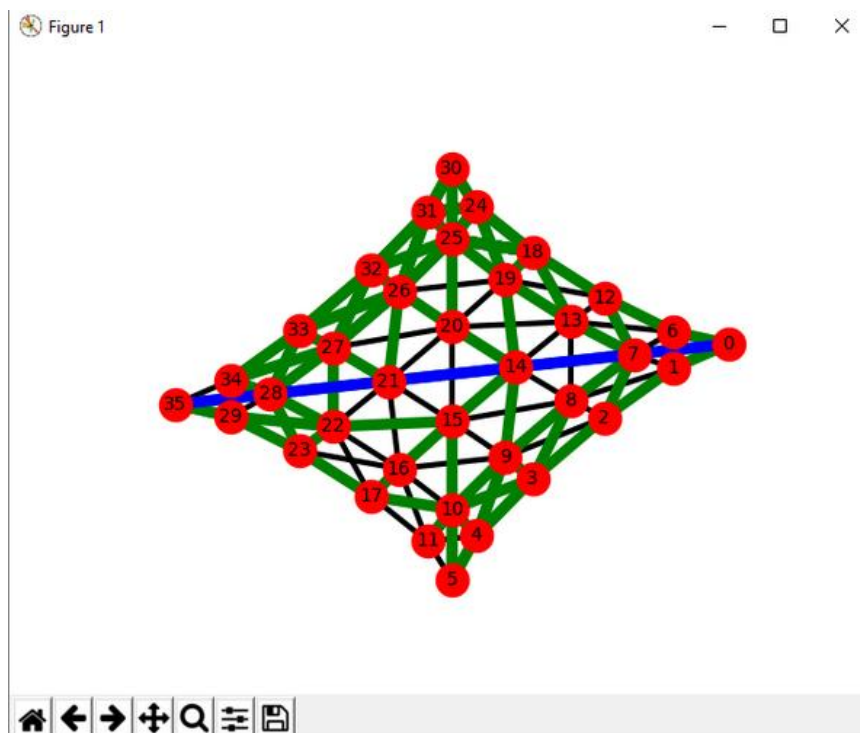
A further improvement to the program would be to enable blitting of the animation. Blitting is a technique which copies bits from one part of a computer's graphical memory to another part (Christophe 2012); optimising animations by only redrawing items which have changed between frames. The inbuilt Matplotlib animation module contains a blitting feature, but my animation method does not; this is causing the animation to render slowly, as on each iteration, it has to redraw the whole node network.

7. Appendix

[1] Initial Matplotlib display of an interconnect set of 5 NetworkX nodes.



[2] The shortest path found using a brute-force search approach.



8. References

- Arun, Amitha. "Pathfinding Algorithms in Multi-Agent Systems." *NCCA Staff Bournemouth University*. 8 2014.
<https://nccastaff.bournemouth.ac.uk/jmacey/MastersProjects/MSc15/01Amitha/thesis.pdf> (accessed 5 19, 2017).
- Christophe, Bessis. *Gamedev Glossary: What Is 'Blitting'?* 26 12 2012.
https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.DiGraph.add_edges_from.html (accessed 5 19, 2017).
- Matplotlib. *animation module - Matplotlib 2.02 Documentation*. 2002.
http://matplotlib.org/api/animation_api.html (accessed 4 20, 2017).
- NetworkX. *add_edges_from - NetworkX 1.9 Documentation*. 2015.
https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.DiGraph.add_edges_from.html (accessed 5 19, 2017).
- . *Creating a Graph - NetworkX 1.9 Documentation*. 2017.
<https://networkx.github.io/documentation/networkx-1.9/tutorial/tutorial.html#node-attributes> (accessed 4 20, 2017).
- . *Weighted Graph - NetworkX 1.9 Documentation*. 2017.
https://networkx.github.io/documentation/networkx-1.9/examples/drawing/weighted_graph.html (accessed 3 10, 2017).
- Patel, Amit. *Heuristics*. 2011. <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html> (accessed 4 28, 2017).
- . *Introduction To A**. 2014.
<http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html> (accessed 3 10, 2017).
- Python. *5. Data Structures - Python 3.5 Documentation*. 2017.
<https://docs.python.org/3.5/tutorial/datastructures.html> (accessed 5 2, 2017).
- StackOverflow. *algorithm- assign x,y coords in networkx/python for a* search heuristic*. 22 4 2017.
<http://stackoverflow.com/questions/43558873/assigning-x-y-coords-in-networkx-python-for-a-search-heuristic/43561707#43561707> (accessed 4 22, 2017).
- . *How do I fix the deprecation warning that comes with pylab.pause?* 7 4 2014.
<http://stackoverflow.com/questions/22873410/how-do-i-fix-the-deprecation-warning-that-comes-with-pylab-pause> (accessed 5 1, 2017).
- . *NetworkX - How to change the shape of the node?* 19 6 2015.
<http://stackoverflow.com/questions/30344592/networkx-how-to-change-the-shape-of-the-node> (accessed 5 1, 2017).

- . *When to use `cla()`, `clf()` or `close()` for clearing a plot in matplotlib*. 19 3 2017. <http://stackoverflow.com/questions/8213522/when-to-use-cla-clf-or-close-for-clearing-a-plot-in-matplotlib> (accessed 5 3, 2017).
- WebPage FX. *ColorPicker by WebPage FX*. 2017. <https://www.webpagefx.com/web-design/color-picker/> (accessed 5 3, 2017).