CS142 Coursework 2: Visualising an algorithm

Chibudom Onuorah
U1709143

Abstract

The purpose of this assignment was to visualise an algorithm using Processing. I chose to visualise Dijkstra's algorithm using the 'Processing' graphical library. I used lab material (most notably lab5) (Warwick, 2019) to create a weighted graph and I implemented Dijkstra's algorithm upon this graph. The data used was a table of connected Underground stations with the distance (in km) between them also listed. The report explains the approaches I took and also gives justifications for certain choices.

Motivations

I decided to visualise Dijkstra's algorithm because I remember how much I enjoyed learning the algorithm during sixth form higher level mathematics. I was taught how to calculate the shortest path between Nodes (via Dijkstra) in a very visual way; other students were taught to draw an adjacency matrix, whereas I was taught to visualise the graph (ie. draw it out) and conduct calculations using the drawn graph. I remembered how useful and intuitive this visual method was, so I decided to see if I could replicate it using Processing. I believe that this is a good motive for a visualisation because it is in line with Edward Tufte's (Tufte, n.d.) ideologies surrounding graphical excellence, one of which state that 'graphical excellence consists of communicating complex ideas with clarity'. My aim is to be able to simply visualise Dijkstra's algorithm.

Research

I built upon skills learned in the previous labs and coursework by doing further exploration into the ControlP5 GUI library. During my research I learned that ControlP5 is a GUI library composed by Andreas Schlegel. The scope of accessible controllers incorporates Slider, Button, Toggle, Knob, Textfield, RadioButton, Checkbox, Lists among others (Sojamo.de, 2019). These controllers can be effectively added to a handling sketch or showed inside a different control window. They can be composed in tabs or gatherings and also rendered into PGraphics cradles. Furthermore, controller state can also be saved to file using JSON formatting. I decided that a button would be the most relevant controller for my visualisation. I planned to use the button as a switch to begin the visualisation.

Furthermore, during my research I read an Article by Mike Bostock (Bostock, 2019) that described the reasoning behind the visualisation of algorithms. I found this particularly useful in determining the purpose of my visualisation. The article stated that the visualisation of algorithms are for four main purposes: entertaining, teaching, debugging and learning. This led me to decide that my algorithm would be used for learning and debugging purposes. My plan was to create a visualisation that could show an individual how Dijkstra's algorithm works and also show them a break-down of steps taken. In hindsight, implementing Dijkstra's algorithm allowed me to find errors in my implementation, despite the output looking correct.

Dataset

In order to successfully implement Dijkstra's algorithm, I needed a dataset that could easily be morphed into a weighted graph. I chose to use a dataset that listed connected underground station and the distances (in km) between them. I envisaged that the underground stations be represented by Nodes, whereas the distances between the stations would be represented by edges, which would be drawn using a straight line. As I continued planning, it became apparent that my visualisation concept combined with my dataset would produce something similar to a tube map. Being able to determine the shortest distance between two nodes (stations) on my graph, would give me the same ability as an application like google maps. I did some research into Google Map's algorithm; I was a little disappointed to realise that they did not use Dijkstra's algorithm but use A* algorithm (Motherboard, 2019)

instead, which is described to be a more efficient algorithm that 'finds the shortest path and alternative routes in real time' (Motherboard, 2019)

NottingHill	Queensway	0.69
Queensway	Lancaster	0.9
Oxford	Holborn	1.46
Holborn	Chancery	0.4
Pauls	Bank	0.74
Pauls	Oxford	3.7
Lancaster	Oxford	3.7
Queensway	Holborn	6.11

Figure 1 – Dataset

What makes a good visualisation?

Discovering Google's use of the A* algorithm led me to re-think my visualisation. I began to realise that Dijkstra's algorithm is not as efficient as other algorithms, so I began considering other more efficient algorithms to visualise.

Dijkstra's algorithm is not as efficient as other algorithms, it stands out because of its capacity to locate the shortest way from one node to each other node. As opposed to simply finding the most limited way from the beginning hub to another particular hub, the calculation attempts to locate the briefest way to each and every reachable hub – gave the chart doesn't change. I remembered the article by Mike Bostock that I had read previously and re-considered some of the purposes of visualisation that he had mentioned. I decided that the purpose of my visualisation would be for informative/educational purposes; it is not meant to show the most efficient algorithm, rather its purpose would be to explain/visualise Dijkstra's algorithm. Furthermore, during lectures, I learnt that a good algorithm visualisation would 'seek to understand the logical rules that describe behaviour'. On the basis of that, I made the decision to create my visualisation purely for informative/education purposes, which greatly influenced my decisions when coding.

Visualisation Processes and Techniques

Planning. Once I had chosen my dataset, I began to create the data structures to store the Nodes and Edges.

I specifically chose to implement the graph from scratch by creating Node and Edge classes instead of importing a graph library. This means that I could have more control over the program and add my own methods to the Node and Edge classes that may not have been available if I had imported a Graph Library to draw the graph for me. Also, my coding the graph from scratch, I would be able to develop a concrete understanding of the implementation of the algorithm.

```
class Node {
  float x, y;
  float dx, dy;
  String label;
  HashMap<Node, Integer> adjacentNodes = new HashMap();
  color currentCol = color(0, 0, 255);
  color original = color(0, 0, 255);
  color pathColor = color(252, 148, 181);

  //constructor
  Node(String label) {
    this.label = label;
    x = random(width);
    y = random(height);
  }
```

Figure 2 – Node class

The Node class (pictured above) was fairly simple to implement; it uses a HashMap to store adjacent nodes and their distances. I specifically chose a HashMap due to the fact that it is easier and faster to retrieve info O(1), and also it can store Nodes and distance within the same data Structure. I envisioned that the use of a HashMap would make the creation of an adjacency table much easier to display.

Processes: Coding

Two important features that I began coding with, was the relax() and update() methods. During lab 5, whilst working on graphs, I discovered the benefit of using force-directed positioning. Using force-directed layout, through a series of calculations, the methods automatically calculate the optimal layout for the graph by what would almost look like trial and error. However, upon closer look, in a force directed layout, edges act like springs with a target length. At each time step, each Edge tries to get its length a little close to its target length. Because several edges are interconnected, the elements push and pull on one another and over time reconcile to a best possible fit.

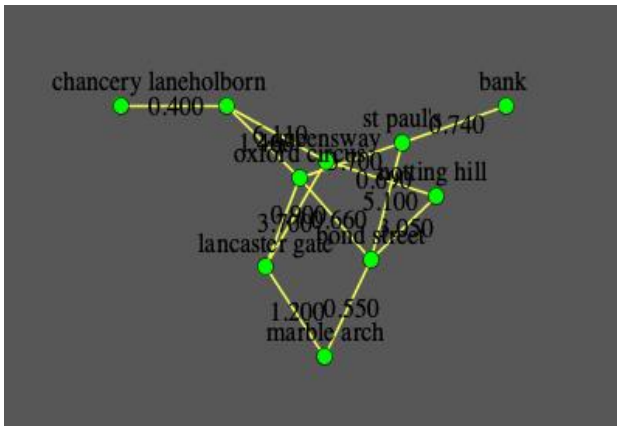

Figure 3 – Non force-directed graph()

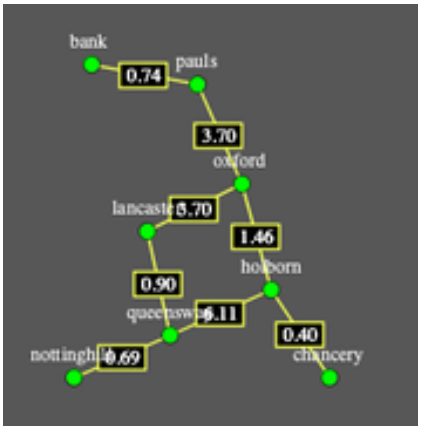

Figure 4 – Force-directed graph

The ‘update’ method is unique because it thinks about the specific Node against every single other Node to watch that none are excessively close. If they are, a slight counterbalance is included. The update() method updates the positioning of the Node is constantly by constricting dx and dy such that they don't surpass five pixels in either heading and then increasing the value of x and y.

```
void update() {
    x += constrain(dx, -5, 5);
    y += constrain(dy, -5, 5);

    x = constrain(x, 0+150, width-150);
    y = constrain(y, 0+150, height-230);

    dx/=2;
    dy/=2;

}
```

Figure 5 – update() method in Node class

Coding: Implementing the Algorithm

To implement the algorithm, I mainly used a series of HashMaps and ArrayLists. Since Processing does not explicitly have the HashSet data structure, I was forced to improvise using an ArrayList but then checking that the object didn't exist before adding it to the ArrayList. To put it simply, I implemented the algorithm by creating two ArrayLists called 'usedNodes' and 'availableNodes'. User then chooses the start and end nodes. Every node is given a temporary distance value. At the beginning, the origin node distance is set to zero set and to Integer.MAX_VALUE; (to represent infinity) for all other nodes. Set the initial node as current. For the chosen, available node, all adjacent nodes are considered and their temporary values from current Node are determined. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. Once all adjacent have been considered, node is added to usedNodes ArrayList and is not considered again. Once the destination Node is reached, the algorithm stops and the path is created,

Coding: Use of CP5 GUI Library

ControlP5 is a 'button-based' **GUI library** written by Andreas Schlegel that allows the algorithm to receive user input. I created buttons for every Node (station) and allowed the user to determine which station they wanted to be the 'origin' and which station they wanted as the 'end' by clicking on the button with the respective station name on it.

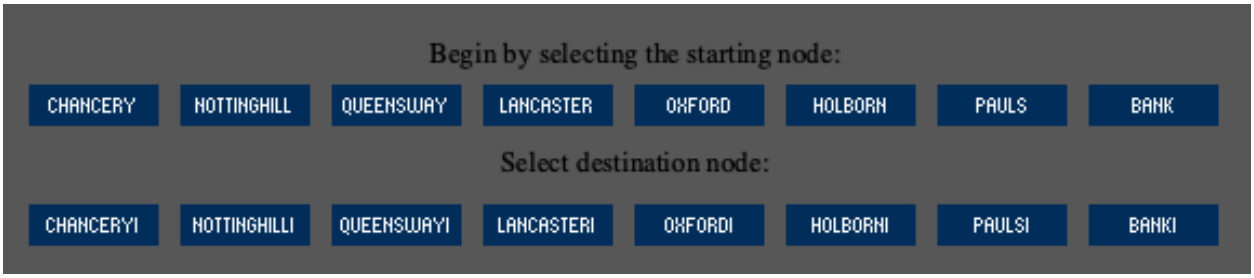

Figure 6 – Station buttons

Figure 7 – Calculate and ShowTree button

I also created a 'Calculate' and 'ShowTree' button. The calculate button begins the execution of Dijkstra's algorithm. This gives the viewer a level of control within the visualisation because the algorithm only begins when they decide for it to. User control was an important part of this visualisation because the purpose of the visualisation is to allow users to learn and debug, so the addition of buttons to increase user control was a deliberate addition.

I created a method/button called showTree() that, once the shortest path is determined, removes the edges of all other Nodes that are not included in the shortest path. By removing the edges from the other nodes, I am isolating yet emphasising the shortest path and allowing the viewer to get a clearer view, without the distractions of the weights/edges of other irrelevant nodes.

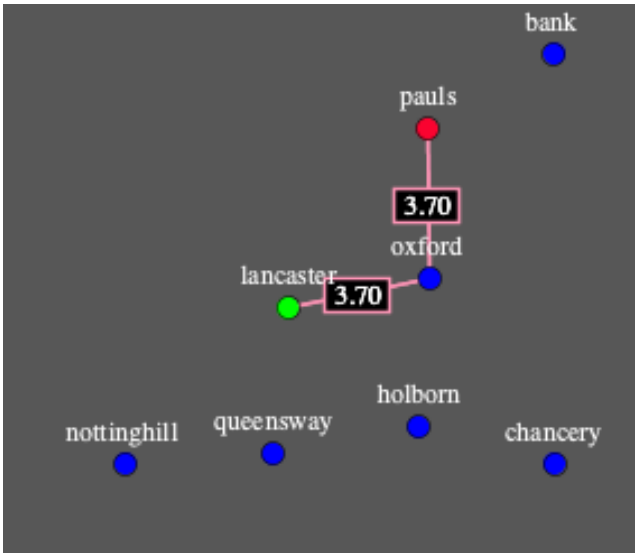

Figure 8 – showTree()

Choices: Colour

As I continued to code, I realised the need for a colour scheme. I chose a grey background as I planned to make the nodes a fairly bright colour, so I wanted there to be a contrast, which would, in turn, emphasise the Nodes more. Similarly, the station names are white such that it contrasts with the background.

How I visualised the shortest path between two edges

By default, the nodes are coloured blue and the edges are coloured yellow. I chose these because they are complementary colours and do not give any misleading interpretations. Since the graph is not directed, once the shortest path is determined, it was not necessarily clear where the starting node is and where the finishing node is. Once the user has chosen the origin Node and the end Node, the origin node's colour is changed to green and the end Node is changed to red. I chose green for the start because is a colour with positive connotations (e.g. green for 'go' in a traffic light) and I chose red for the destination Node because it has 'negative' connotations (e.g. stop signs tend to be red).

I decided to change the colour of the edges inside the shortest path, to signify that they had been 'chosen' by Dijkstra's algorithm. I created a changeColour() method in the Edge class that can quickly alternate the colour of an edge between yellow (default colour) and red, such that it appears that the edge is 'flashing' red. I called this method for Edges that were determined to be in that shortest path.

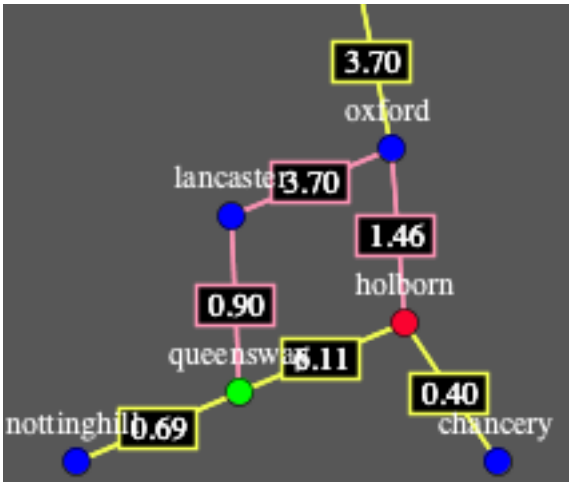

Figure 9 – Colour coded nodes

Choices: Adjacency Table

In line with my decision to create this visualisation for debugging purposes, I decided that it would be useful to see the adjacency table in order to allow the viewer to see the final distances between their chosen 'origin' node and all the other nodes. This means that if they were to try and implement Dijkstra by themselves, on paper, for example, they would be able to quickly check if their answer is correct without having to run the visualisation again. Therefore, the adjacency table can be used as reference to debug or to check if an answer is correct. Also, when the user chooses the origin node, the right table heading 'distance from [originNode]' is updated for greater clarity.

Node	Dist from: queensway
pauls	8.30 km
lancaster	0.90 km
bank	9.04 km
chancery	6.46 km
queensway	0.00 km
holborn	6.06 km
nottinghill	0.69 km
oxford	4.60 km

Figure 10 – Adjacency table

U1709143 CHIBUDOM ONUORAH

Evaluation:

Accuracy:

I ran extensive tests to check that Dijkstra's algorithm was being implemented correctly and I am confident that the shortest distance between any combination of nodes is calculated accurately. However, I did not realise how difficult it would be to implement Dijkstra's algorithm. In hindsight, I spent a great deal of time implementing the code and trying to conceptualize it. If I had chosen an easier algorithm, I could have spent more time on the visualisation of the algorithm rather than writing the algorithm itself.

Purpose:

This visualisation is able to calculate the shortest distance between two nodes based on user entry and display the path in two different layouts. Furthermore, the adjacency list shows the user the distances from the chosen origin Node which allows them to do their own calculations (to other Nodes) and check against the adjacency list to see if their own calculations are correct. This means that the visualisation serves the purpose of teaching and also debugging.

Aesthetically pleasing:

I used a simple blue colour scheme that helps to create a nice interface but does not distract from the graph.

Improvements:

My visualisation would have been better if I were able to explicitly show the comparison process. When the algorithm is checking the distance of adjacent nodes, I would have been good to make those nodes to flash to show the user which Nodes are being considered and then which Node is eventually chosen. The issue I had with implementing this is that, in order to show this sort of step-by-step process, I would have had to decrease the framerate further, which would have further slowed down the visualisation

Appendix

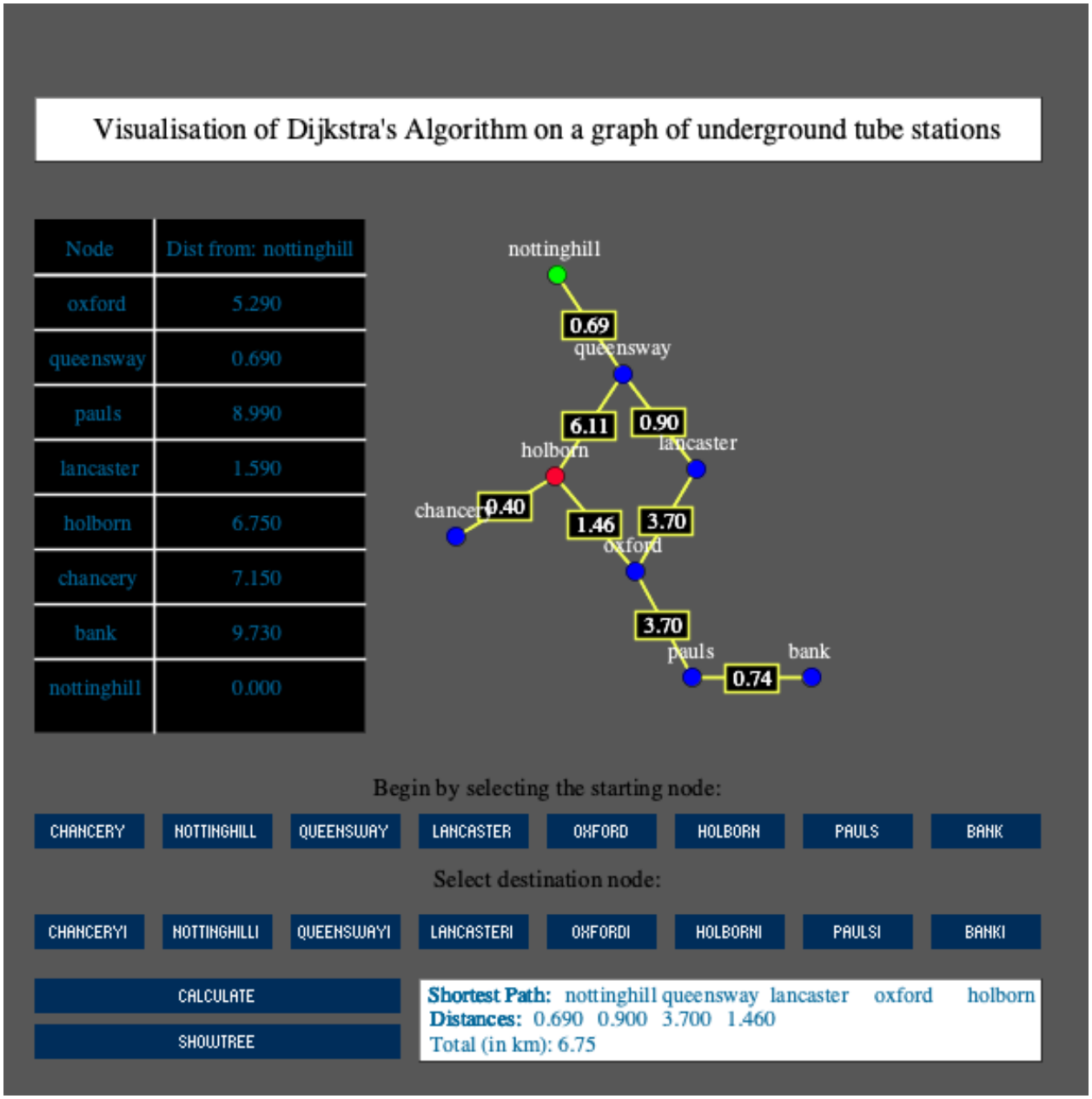

Figure 9 – Final Version

References:

Anon, (2019). *Quora*. [online] Available at: https://www.quora.com/How-does-the-algorithm-of-Google-Maps-work [Accessed 29 Mar. 2019].

Bostock, M. (2019). *Visualizing Algorithms*. [online] Bost.ocks.org. Available at: https://bost.ocks.org/mike/algorithms/ [Accessed 29 Mar. 2019].

En.wikipedia.org. (2019). *Dijkstra's algorithm*. [online] Available at: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm [Accessed 29 Mar. 2019].

Motherboard. (2019). *The Simple, Elegant Algorithm That Makes Google Maps Possible*. [online] Available at: https://motherboard.vice.com/en_us/article/4x3pp9/the-simple-elegant-algorithm-that-makes-google-maps-possible [Accessed 29 Mar. 2019].

Myrouteonline.com. (2019). *What's the Best Shortest Path Algorithm | MyRouteOnline*. [online] Available at: https://www.myrouteonline.com/blog/what-is-the-best-shortest-path-algorithm [Accessed 29 Mar. 2019].

Sojamo.de. (2019). [online] Available at: http://www.sojamo.de/libraries/controlP5/examples/controllers/ControlP5button/ControlP5button.pde [Accessed 29 Mar. 2019].

Tufte, E. (n.d.). *The visual display of quantitative information*.