CS142 Coursework II

Artemy Bulavin
1812276

Introduction

The aim of this visualisation is to demonstrate the process of Dijkstra's Shortest Path algorithm using a simple graph with weighted edges, where colours of vertices and edges change based on the different stages of the algorithm.

Dijkstra's Shortest Path Algorithm

Dijkstra's Shortest Path algorithm is a method for finding the shortest distance from a given source node to all other nodes in a simple, weighted graph [2]. The version of the algorithm used in this visualisation is the original version, as opposed to min-priority queue implementation [3].

The algorithm works as follows:

1. Mark all nodes as unvisited and construct a set of all unvisited nodes.
2. Give the source node a tentative distance of 0, and all other nodes a tentative distance of ∞.
3. For the current node, calculate the tentative distance from the current node to its adjacent neighbours. Compare this distance to the distance already assigned to the node and choose the minimum of the two to be the node's tentative distance.
4. Once all neighbours of current node have been considered, mark it as visited and remove it from the unvisited set. Visited nodes are not visited again.
5. If the unvisited set is empty, the algorithm ends.
6. Choose the next current node from the unvisited set to be the one with the smallest tentative distance. Go to step 3.

The algorithm is implemented using the following pseudocode [6].

```
function DIJKSTRA(Graph G, source)
    Create vertex set Q from G
    for all Vertex v in Q do
        distance[v] ← ∞
        previous[v] ← null
    end for
    distance[source] ← 0
    while Q is not empty do
        u ← vertex with smallest distance[u]
        Remove u from Q
        for all neighbours v of u do
            newDistance ← dist[v] + length(u, v)
            if newDistance < dist[v] then
                dist[v] ← newDistance
                previous[v] ← u
            end if
        end for
    end while
return distance, previous
end function
```

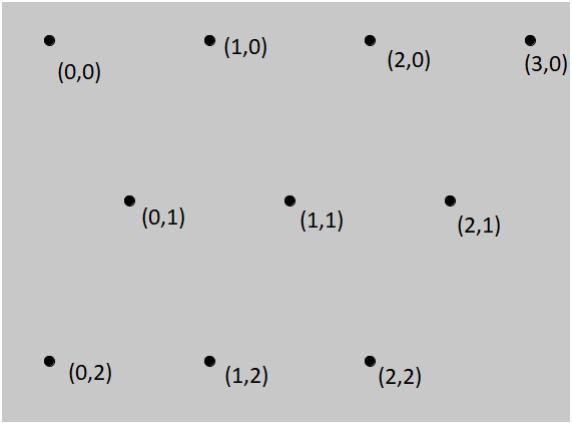
Design and Implementation

The algorithm would be visualised using a simple, connected graph with weighted edges that are positioned isometrically so that the distances between adjacent nodes are equal for all nodes. This is so that edges between nodes are not tangled (especially for larger graphs) and the graph is visually penetrable [1]. Since the nodes are arbitrary their position in the space does not convey any information. This means the system by which the nodes are placed in the space needs to be decided [4]. In order to achieve the best layout and allow for maximum visual penetration, isometric spacing was chosen.

Drawing Vertices

In order to achieve this, `Node` and `Edge` classes were created that would form the structure of the graph. These were encapsulated further by a `Graph` class that would handle graph generation and drawing of the nodes and edges.
Each vertex of the graph contains a `PVector` representing the location of the vertex. During `generateGraph()` in class `Graph`, n vertices are created, where n is the number of points. Each vertex is given an x and y value that ensure isometric positioning, For each vertex a `button` object is also created using the ControlP5 class to allow for interaction later.
The space between each vertex is ultimately determined by the variable `scale`, which is calculated from the number of points and screen dimensions to ensure that for any number of points, the graph scales to fit the space.
This allowed for any number of points to be drawn as an isometric grid (Figure 1).

Figure 1: Initial Vertex positioning



Drawing Edges

Next, the edges would have to be drawn between each node. To do this, a matrix representation of the graph is created using a 2D array that assigns each node to a row and column (Figure 1). This allowed for rules to be established that determined how the nodes would be connected to each other.
For a node in the matrix with position (i, j) where i is the column and j is the row:

- If a node is in the top row i.e $j = 0$ then create an edge to $(i + 1, j)$ and $(i, j + 1)$.
- If a node is on the bottom row and is on an *even* row i.e $j \pmod 2 \equiv 0$, then create an edge to $(i + 1, j)$ and $(i, j - 1)$. If it's on an odd row, create an edge to $(i + 1, j - 1)$ and $(i + 1, j)$.
- Otherwise, if the vertex is not on the top or bottom row, and is on an even row then create an edge to $(i, j - 1)$, $(i, j + 1)$ and $(i + 1, j)$. If it's on an odd row, then create edges to $(i + 1, j - 1)$, $(i + 1, j)$ and $(i + 1, j + 1)$.

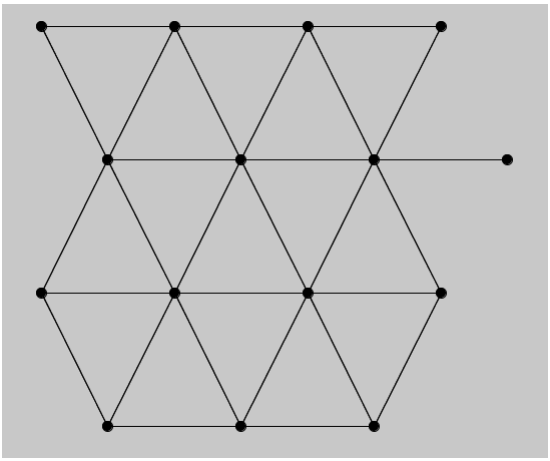
Each `Edge` object created is also assigned a random weight between 30 and 110 inclusive. This is so that there is enough variation in the weights of the edges for the algorithm to illustrate paths between nodes that were not intuitive or apparent immediately.
This allowed for an almost complete graph to be drawn, with drawn edges on the screen corresponding to edges in the Graph data structure (Figure 2).

Labelling

In order to label vertices, each vertex has a letter A-Z assigned to it that is drawn directly above. While the labels are redundant in the sense that they are not necessary for the algorithm, their change in colour of the labels helps to illustrate the current stage of the algorithm. Vertices are also coloured based on whether they are the current vertex, have been visited or remain unvisited. This is done using the `State` enumeration. The current vertex is drawn in yellow, visited vertices and edges in green and unvisited vertices in magenta purple. The colours are chosen to accommodate for red-green colour-blindness - the yellow provides enough distinction from the magenta and green, and the magenta will not appear similar to the green [7].

In order to label edges, the weight of the edge is drawn by the line as a number. While Mazza [4] suggests to use line thickness/weight or labels for edge weight, line thickness is a pre-attentive attribute that conveys quantitative information poorly in this case. For example, two edge, one

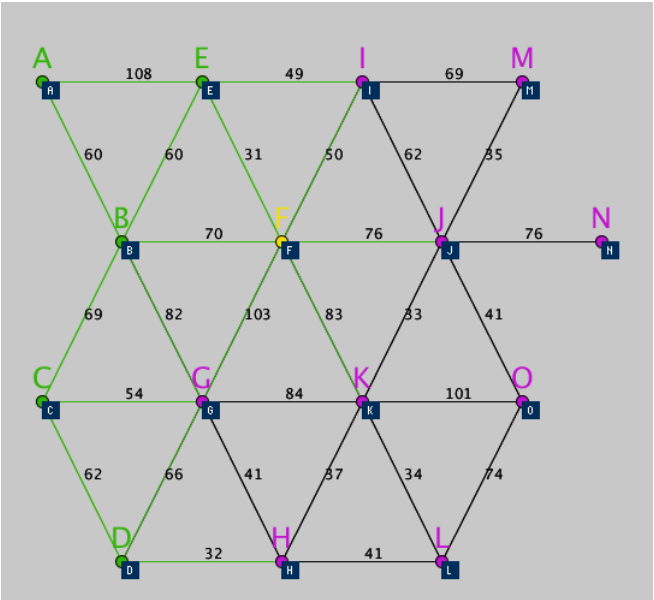
Figure 2: Edges Added



with eight 63 and another with 70 may appear almost identical to the viewer, meaning that once the resulting shortest path is displayed it cannot easily be verified by checking the weights of the edges in the path. Therefore, while adding extra text to the visualisation may require the viewer to mentally maintain numbers in their working memory and draw attention away from the algorithm [5], this is preferred for when the algorithm ends and the viewer can interact with and verify the shortest paths themselves.

With the added labels and colours, the algorithm is ready to be animated (Figure 3).

Figure 3: Labels Added



Animation

References

[1] Card, S. K., Mackinlay J. D., Shneiderman, B. *Readings in information visualization: using vision to think*. San Francisco: Morgan Kaufmann, (1999).

[2] Djikstra, E. W. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, (1959).

[3] Fredman, L. M., Tarjan, E. R. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, (1987).

[4] Mazza, R. *Introduction to information visualization*. London: Springer Science & Business Media, (2009).

[5] E. R. Tufte. *The Visual Display of Quantitative Information (2nd ed.)*. Cheshire, CT: Graphics Press, (2001).

[6] Wikipedia contributors. Dijkstra's algorithm — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=888813153`, 2019. [Online; accessed 27-March-2019].

[7] Wong, B. *Points of view: Color blindness*. Nature Publishing Group, (2011). `https://www.nature.com/articles/nmeth.1618` [Online; accessed 27-March-2019].