**Project 3 Report – Bret Kagebein, Justin Morgan, Gavin Smith**

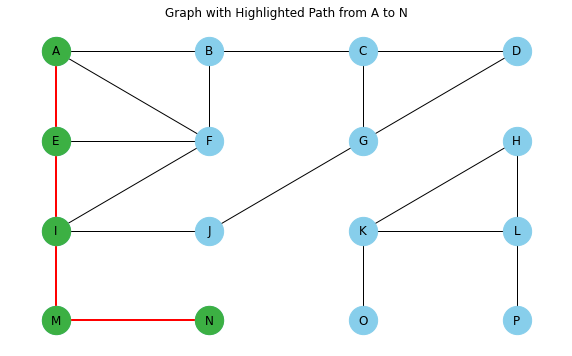
1a) Yes, both DFS and BFS can find all connected components of an undirected graph.

Logically, DFS will keep going as far as it can down one path until there are no more unvisited nodes (example from the graph, [A, B, C, D, G] – or, alternatively, [A, B, F, E, I, J, G, C, D]). Then, after the visited node, it will backtrack, and check visited nodes for paths leading to unvisited nodes until it finds one (backtrack to ‘B’ finding ’F’ in the first case and backtrack to ‘I’ finding ‘M’ in the second case), then continues this process until you backtrack all the way to the first node with no other unexplored path.

BFS will search every single node for its adjacent vertices and list all previously unseen connected nodes (example from the graph, it will list A, then all of its connected nodes – [A, B, F, E,] , then search B’s connected nodes and add any unlisted ones – [C] – then F’s nodes – [I] – then go to E’s and find nothing new, then go back to B’s first connected node C and add any new unlisted nodes – [D, G], then search F’s connected node I, etc.

1b) As both DFS and BFS can find all connected components in a graph, it can naturally determine if there is a path between two nodes.

1c) However, while DFS and BFS can both find paths, the two paths are not always the same. This is because, while BFS checks every node and can always find (one of) the shortest path(s), DFS prioritizes topographical order. In the case of our program, the vertices are listed in alphabetical order, so DFS will prioritize a path that goes in alphabetical order.

So, using the graph above, if you ask BFS to find a path a path from A to N, it will give you [A, E, I, M, N] – a length of 5, the shortest possible path. 

However, DFS will prioritize alphabetical order, so the path it will give is [A, B, C, D, G, J, I, M, N], which has a length of 8 and is one the most inefficient paths from A to N.

A diagram of a path

Description automatically generated

2a) The application (a version of the scc.py file provided with a few of the timer applications and some comments omitted) suggested that the strongly connected components are:

[11], [12], [3, 2, 1], [4], [9, 5, 6, 7, 10, 8]

2b)

A drawing of a diagram on a lined paper

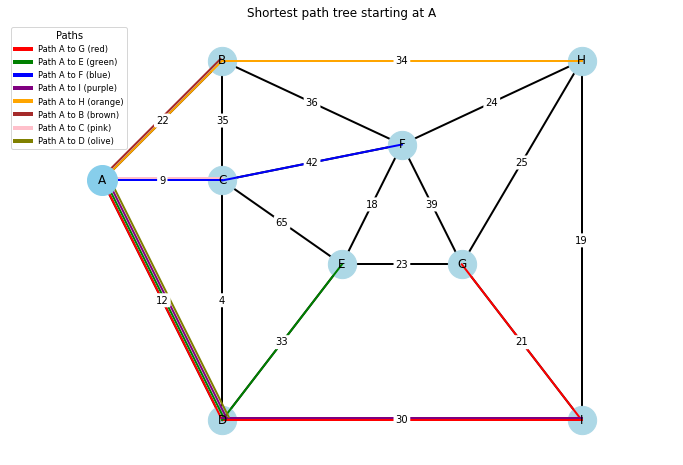
Description automatically generated

2c)

A diagram of a flowchart

Description automatically generated

3a)



Shortest paths from node A:

(Length Order) (Alphabetical Order)

Node A has a path of length 0: ['A'] Node A has a path of length 0: ['A']

Node C has a path of length 9: ['A', 'C'] Node B has a path of length 22: ['A', 'B']

Node D has a path of length 12: ['A', 'D'] Node C has a path of length 9: ['A', 'C']

Node B has a path of length 22: ['A', 'B'] Node D has a path of length 12: ['A', 'D']

Node I has a path of length 42: ['A', 'D', 'I'] Node E has a path of length 45: ['A', 'D', 'E']

Node E has a path of length 45: ['A', 'D', 'E'] Node F has a path of length 51: ['A', 'C', 'F']

Node F has a path of length 51: ['A', 'C', 'F'] Node G has a path of length 63: ['A', 'D', 'I', 'G']

Node H has a path of length 56: ['A', 'B', 'H'] Node H has a path of length 56: ['A', 'B', 'H']

Node G has a path of length 63: ['A', 'D', 'I', 'G'] Node I has a path of length 42: ['A', 'D', 'I']

3b) Our code used Dijkstra’s Algorithm and came up with an MST that closely resembled the one I found by hand (I suspect the output is in alphabetical order and not chosen order):

A diagram of a tree

Description automatically generated

By Hand (Path Order): By Program (Alphabetical Order):

Edge (A, C) with weight: 9 Edge (A, C) with weight: 9

Edge (C, D) with weight:4 Edge (A, B) with weight: 22

Edge (A, B) with weight: 22 Edge (C, D) with weight: 4

Edge (D, I) with weight: 30 Edge (D, I) with weight: 30

Edge (I, H) with weight: 19 Edge (E, F) with weight: 18

Edge (I, G) with weight: 21 Edge (E, G) with weight: 23

Edge (G, E) with weight: 23 Edge (G, I) with weight: 21

Edge (E, F) with weight: 18 Edge (H, I) with weight: 19

Total weight of MST: 146 Total weight of MST: 146

3c) A Shortest Path Tree and a Minimum Spanning Tree are completely different things. A Shortest Path Tree (SPT) shows you the shortest paths between a starting node and any other node (see ‘1C’ where ‘A’ was the starting node), whereas a Minimum Spanning Tree (MST) is a representation of a graph that shows only the shortest connections between each node and has the smallest overall weight. The SPTs purpose is to show all of the smallest branches from one centralized point of interest, whereas the MST shows the smallest overall graph.

Example, when measuring the shortest path from A to I in both cases:

SPT: [A, D, I] – Length 42 MST: [A, C, D, I] – Length 43

But, comparing the total weight of all vertices for both graphs:

SPT: 170 MST: 146

So, SPTs and MSTs are fundamentally different and have their own purpose.

3d) No, if a graph has an edge with a negative weight, you cannot apply Dijkstra’s algorithm for several reasons. First, the algorithm cannot account for large negative numbers down the line that could conceivably make taking longer paths more advantageous. The best way to handle graphs with negative edges is to treat them like directed graphs, which can be better handled with something like the Bellman-Ford algorithm.

CREDITS:

Justin Morgan – Implemented code required for problems 1 and 3

Bret Kagebein – Implemented code required for problem 2 and wrote report

Gavin Smith – Troubleshooting for code and proofread report

RESOURCES USED:

scc.py code provided in the class Blackboard:

(the following citation is from the comment at the beginning of the code)

#https://github.com/ChuntaoLu/Algorithms-Design-and-

#Analysis/blob/master/week4%20Graph%20search%20and%20SCC/scc.py#L107

NetworkX citing:

Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics, and function using NetworkX”, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

<https://networkx.org/documentation/stable/reference/drawing.html> <https://networkx.org/documentation/stable/reference/algorithms/traversal.html>

matplotlib: <https://matplotlib.org/stable/tutorials/pyplot.html> <https://matplotlib.org/stable/index.html>