Introduction

Graphs are non-linear data structures consisting of a collection of nodes with edges connecting amongst them. Road networks, which are vast and complex, are often represented graphically, with nodes being intersections/endpoints and edges being roads connecting them together. These data structures could then be parsed into applications such as Google maps to provide directions. With applications like these, the algorithms would strive to find the best and most effective path to get to their destination. In this project, we are given an undirected unweighted graph with *n* number of nodes (intersections/endpoint) and *m* number of edges (roads), where there would be *h* number of target nodes (hospital) that we are interested in finding *k* number of paths to.

Aim

The aim of this project is to explore graph traversing algorithms and modify them so as to be able to meet our goals in the most effective manner. Theoretical and empirical study would also be done where we focus on the effects of *h* and *k* on the performance of various algorithms designed.

Resources

Graphs are all sourced from Stanford Network Analysis Project (SNAP):

File name	Size	Number of Nodes	Number of Edges
ROADNET-TX.TXT	57,847 KB	1,379,917	3,843,320
ROADNET-PA.TXT	45,021 KB	1,088,092	3,083,796
ROADNET-CA.TXT	85,730 KB	1,965,206	5,533,214

Upon further analysis and testing, we discovered that the road network graphs are unconnected graphs. While unexpected, our team worked to rectify this issue to the best of our abilities. Our solutions will be elaborated further in this report.

Problems to Solve

1) Design an algorithm for computing the distance from each node in the graph to the nearest target node, output the distance and shortest path for each node to a file.
2) Design an algorithm where the time complexity should not depend on the number of hospitals.
3) Design an algorithm where we can find the top-2 nearest hospitals from each node, in the event of major disasters (i.e fires).
4) Propose an algorithm that works generally for computing the distance from each node to the top-k nearest hospital for any input of *k*.

At first glance, the Multi Source Single Path and Multi Source Multi Path Breadth First Search (BFS) Algorithm should be able to answer the questions above. However, it is unsure if the algorithm's complexities would depend on the number of hospitals, which will be something we will look at in the empirical study section.

Assumptions

For multi-sourced algorithms, besides looking for the nearest hospital, we will also look for the nearest adjacent hospital, in the event where we need to get from the target hospital to the next nearest due to certain circumstances (i.e.shortage of beds). For all algorithms besides the Multi Source Multi Path algorithm, we assume that it is a fully connected graph.

Algorithms Explained

Single Source Single Path BFS

This is the standard BFS algorithm. It visits and marks all the key nodes in a graph in a breadthwise fashion. From a specific starting node, it would visit all the nodes adjacent to it. These adjacent nodes will be accessed one by one. Once the algorithm analyses the starting node, it would then move towards the nearest unvisited nodes and analyse them. This would continue until a target node has been reached, which then gives us the shortest path. It would be considered the shortest path as it was the first to be found. This would then be run for all the nodes in the graph.

Single Source Multi Path BFS

This algorithm is a modification of the Single Source BFS to find more than one path. For every source node, we have added a results list of *k* length which would store the found paths to *h*. The algorithm will run until the number of paths that is required is found, after which the results would be returned.

Multi Source Single Path BFS

In this algorithm, we start searching from the target nodes instead of a single source node. This is implemented by first creating a dictionary to keep track of the nodes that have been visited. Next, we create a tree with a node called the "master node", where it would aid in holding all the results, and determine the paths. Then, all the target nodes are pushed into the tree. For example, if [5, 10, 15, 20] are the target nodes, they would all become "leaves" to the master node as they are the first level of the tree. The default BFS would then be done in turns for the leaves in the tree. In our example it would be nodes [5, 10, 15, 20], and when a node is reached, it would be reflected in the dictionary. This would be done until all nodes in the graph are found.

Multi Source Multi Path BFS

Here, we further modified the multi source algorithm to be able to find multiple paths. Instead of having one tree containing all the target nodes, a tree is created for each target node. A dictionary would be used to keep track of the statistics of the current search. For example, given target nodes [5, 10, 15], there would be 3 trees created with 5, 10, 15 being the root node. For each tree, BFS would then be performed iteratively. Every time a node is added into the tree, it would be updated into the dictionary. For a fully connected graph, the searching is considered complete when there are no more pending nodes. However, if a graph is not fully connected, edges will be added to resolve this issue. This will be elaborated further below.

Significance of our Code

For the Multi Source Multi Path algorithm, we strove to fix the issue of the road network graphs being unconnected by adding edges to the isolated nodes. Essentially, this would be akin to us adding roads between towns to ensure that all towns have access to hospitals.

The detection is done after the initial round of BFS. By looking at the dictionary that keeps track of the current statistics of the search, we are able to detect if the graph is fully connected or not. If it is, then no further actions will be performed. If it is not, we will first look through the nodes to find the unconnected node. Next, we will look through the nodes which we have found paths for and add edges to nodes with only one edge. This mimics us trying to find the most outer nodes in the graph, then adding roads between towns to ensure that all the towns have access to hospitals. This answers the question of having to find paths to target nodes for all nodes in the graph.

Overview of Algorithms Complexities

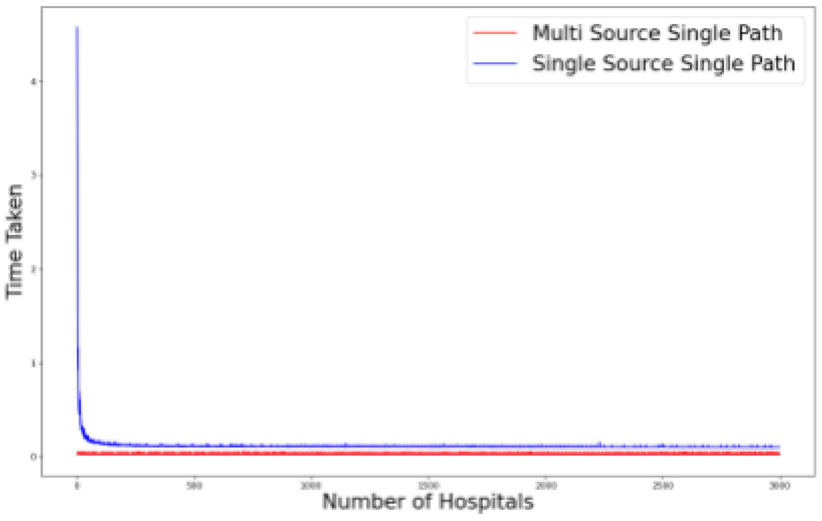
Single Source Single Path BFS	
Time Complexity	Each edge is processed once in the while loop for a total cost of Θ(|E|), each vertex is queued and dequeued once, for a total cost of Θ(|V|). Since adjacency lists are used in the algorithm, the worst time complexity is **Θ(|V|*|E|)**.
Space Complexity	Since all vertices need to be held in the lists and 2 lists are used, the worst space complexity for the algorithm is **Θ(2*|V|)**.
Usage	2 lists for every node

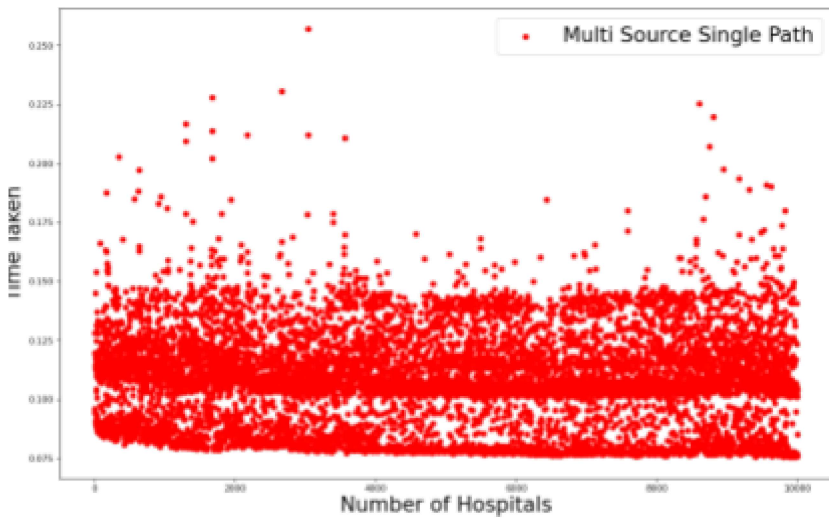
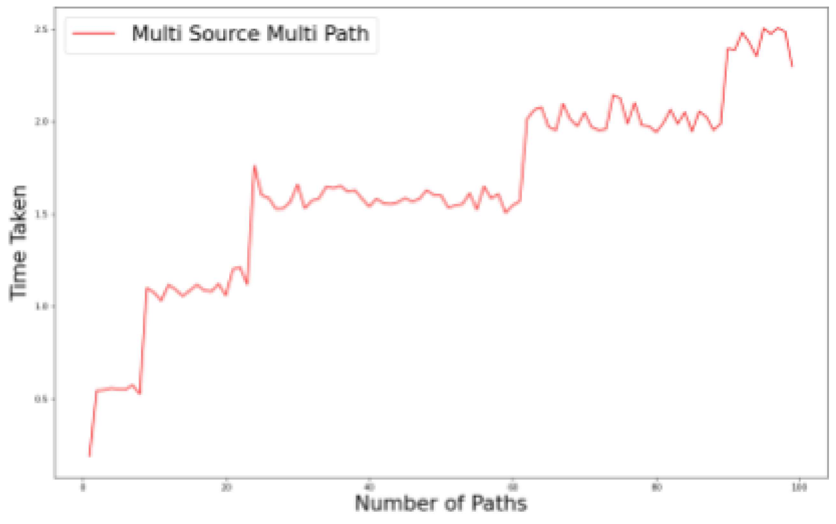
Single Source Multi Path BFS	
Time Complexity	In this case, since the algorithm keeps running (performing BFS) until the required number of paths are found, the worst time complexity is **Θ(k*(|V|*|E|))**, with *k* being the number of required paths.
Space Complexity	Since all vertices need to be held in the lists and 3 lists are used, the worst space complexity for the algorithm is **Θ(3.|V|)**.
Usage	3 lists for every node

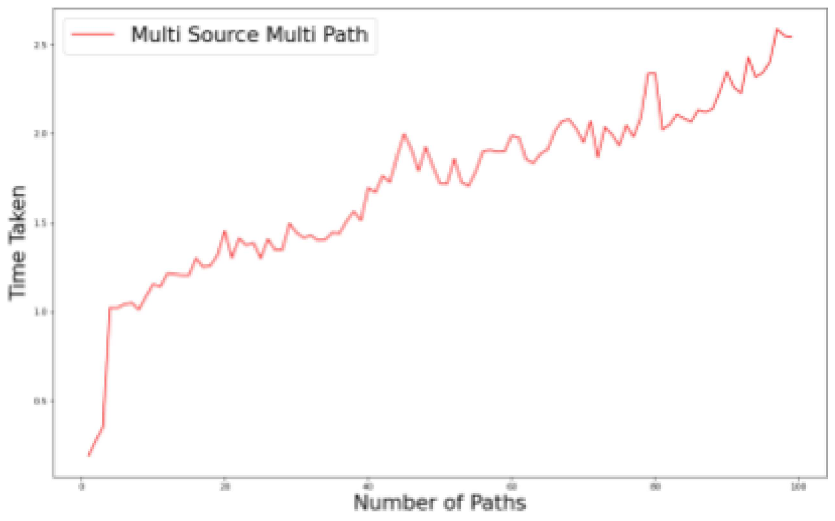
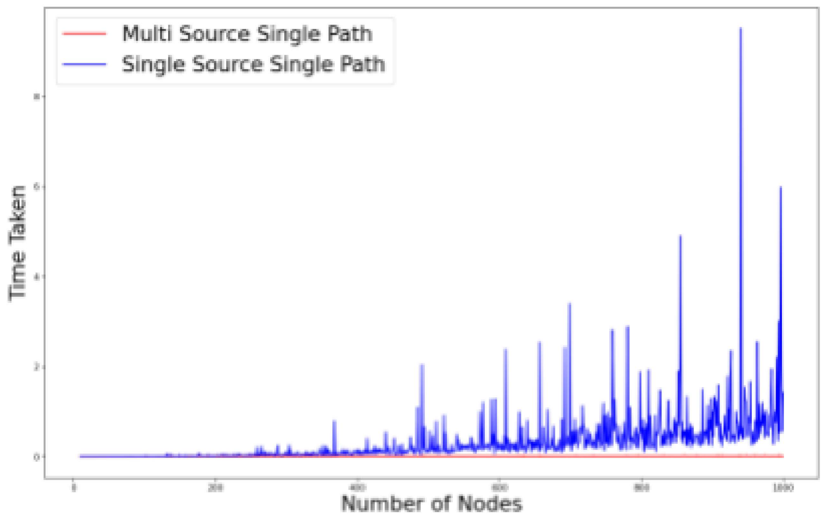
Multi Source Single Path BFS	
Time Complexity	Each edge is processed and each vertex is put into the tree. Hence, the time complexity is Θ(|V|+|E|). The code executes the same way regardless of graph structure, hence, both best and worst case have the same time complexity of **Θ(|V|+|E|)**.
Space Complexity	The space complexity for this algorithm is the same as the basic (single source single path) BFS, **Θ(2*|V|)**, since all vertices are visited and stored in a tree and dictionary.
Usage	● 1 dictionary for 1 search ● 1 tree for 1 search

Multi Source Multi Path BFS	
Time Complexity	The time complexity of the multi source multi path BFS can be derived from the multi source single path BFS. In this case, it is **Θ(k*(|V|+|E|))**, where k represents the number of paths that need to be found.
Space Complexity	Since all vertices need to be held and (h+1) dictionary and trees are used, the worst space complexity for the algorithm is **Θ((h+1)*|V|)**.
Usage	● Uses 1 dictionary for 1 search done ● Uses h trees for 1 search done → where h is the number of hospitals

Empirical Study

Effects of the number of hospitals (*h*)	 *Figure 1: Comparison of time taken for h from 0 - 3000*

	 *Figure 2: Closer look at the time taken for Multi Source Single Path. We can conclude that its time complexity will not be affected by the number of hospitals*
Effects of the number of paths needed to be found (*k*)	 *Figure 3: Time taken as the number of paths to be found increases. Lesser checks are performed in this configuration, and we would recommend this if the number of paths to be found is below 10*

	 *Figure 4: In this configuration, more checks are performed. While the time taken is increased, it is generally more consistent when compared to the "staircase" observed in Figure 3. For this, we would recommend it if the number of paths to be found is above 10*
Effects of the number of nodes (*n*)	 *Figure 5: The time taken for Multi Source Single Path increases at a much slower rate when compared to Single Source Single Path, making it look like it is not increasing at all*

Analysis

From Figure 1, we can observe that the Multi Source algorithm is generally less affected by the number of 0hospitals when compared against the Single Source algorithm. A possible reason could be that in the Multi Source algorithm, the target nodes constitute the first level of the trees, and any effects the number of hospitals might have would only be contained within that level. This also aligns with our theoretical analysis, where the time complexity for Single Source Single Path is **Θ(|V|*|E|)**, while Multi Source Single Path is **Θ(|V|+|E|)**.

Conclusion

In conclusion, we can observe that there is not a *one-size-fits-all* algorithm. However, we can notice that the Multi Source algorithms tend to outperform the Single Source ones. In our given scenario, if we were only needed to find a single path, we would recommend the **Multi Source Single Path** algorithm. Whereas if we needed to find the path of each node to *top-k* hospitals for a given input of *k*, we would instead recommend the **Multi Source Multi Path** algorithm.

References

1. J. Yung, "Python lists VS dictionaries: The space-time tradeoff," *Jessica Yung*, 29-Sep-2018. [Online]. Available: https://www.jessicayung.com/python-lists-vs-dictionaries-the-space-time-tradeoff/ [Accessed: 29-Oct-2021].
2. T. Birchard, "Visualize folder structures with Python's Treelib," *Hackers and Slackers*, 23-Jun-2020. [Online]. Available: https://hackersandslackers.com/python-tree-hierachies-treelib/ [Accessed: 29-Oct-2021].