Network Routing Protocol using Data Structures and Algorithms to solve Maximum-Bandwidth Path Problem

Introduction:

The aim of this project is to find the s-t path of the largest bandwidth for a random undirected and weighted graph from vertex s to vertex t. The followings are the major steps in this project:

- Random Graph Generation:
1. Graph 1, the average vertex degree is 6.
2. Graph 2, each vertex is adjacent to about 20% of the other vertices.
3. Each edge in both the graphs have a positive weight.
- Heap Structure: Implementing the subroutines for Maximum, Insert, and Delete.
- Routing Algorithms:
1. Dijkstra's algorithm without using a heap structure.
2. Dijkstra's algorithm using a heap structure for fringes.
3. Kruskal's algorithm where the edges are sorted by HeapSort.
- Testing: Test routing algorithms on 5 pairs of sparse G1 and dense G2 graphs. 5 pairs of source and destination vertices are selected at random and the time taken by each algorithm is to be recorded.
- Report

Random Graph Generation:

Here I've used the python function erdos_renyi_graph() from the library networkx.generators.random_graphs to generate both type of required graphs. g1: Here the graph contains 5000 vertices and approximately 15000 edges (that is, degree 6 = 6 * 5000 / 2).
g2: Here the graph contains 5000 vertices and approximately 2500000 edges (that is, approximately each vertex is adjacent to 20% of the vertices).
For both the graphs a random number generator [function random.randint()] is used to randomly generate integer weights of the edges from 1 to 100 (both inclusive).

Heap Structure:

Here, we've created a Max Heap Structure to store, and access our graphs. In this structure the maximum element is the root node of the graph and all the elements under each node are smaller than that particular node. In our Heap Structure, we've created the functions Search, Maximum, Delete, and Insert.

The Maximum function returns the maximum value in the graph (the root node).
The Delete function first uses to the search function to search the value to be deleted and then deletes it, and re-arranges the heap so that the properties of Max Heap is retained.
The Insert function first inserts a value at the required location and then re-arranges the heap so that the properties of Max Heap is retained.

Routing Algorithms:

The first algorithm used is the Dijkstra's algorithm without using a heap structure. Dijkstra's algorithm always finds an optimal solution when all the edge weights are positive. This algorithm has a time complexity of O (n_2). Since the time complexity is not dependent on the number of edges, hence we do not expect a significant change in time when the algorithm is run on a sparse or a dense graph. But since the algorithms ends up iterating over higher number of edges in case of the dense graph, hence the algorithm has a higher runtime on denser graph compared to sparse graph.

The second algorithm used is the Dijkstra's algorithm using a heap structure. Dijkstra's algorithm always finds an optimal solution when all the edge weights are positive. This algorithm has a time complexity of O ((n + m) log n). Since the time complexity is dependent on both the number of vertices and the number of edges, hence we expect an increase in runtime in case of a dense graph compared to the runtime in case of a sparse graph. In case of a full / complete graph (where all nodes are connected with each other), one can expect Dijkstra's Algorithm with Heap Structure to take higher time compared to the Dijkstra's Algorithm without Heap Structure.

The third algorithm used is the Kruskal's Maximum Spanning Tree Algorithm. This algorithm has a time complexity of O (m (1 + log m)). Since the time complexity is dependent on the number of edges, hence it is expected that this algorithm would have a high runtime for a dense graph compared to a sparse graph. This algorithm has two paths, constructing the Maximum Spanning Tree and finding the path. Finding a path takes linear time.

For recording time, the python code "%%time" can be used. For constructing MaxHeap the user defined methods HeapBuild() and MaxHeap() are used. For inserting, deleting, search, and maximum operations the user defined methods insert(), Delete(), search(), and maximum() are used.

Testing:

The Time Complexity of each algorithm is as follows:

Algorithm	Time Complexity
Dijkstra without Heap	O (n_2)
Dijkstra with Heap	O ((n + m) log n)
Maximum Spanning Tree (Kruskal's Algorithm)	O (m (1 + log m))

Conclusions:

Case of Degree 6 graph: In this case the number of edges is proportional to the number of vertices, that is, m = degree * n / 2 = 3 * n. This is an example of the Sparse Graph.

Dijkstra's algorithm with heap is faster compared to Dijkstra's algorithm without heap. This happens simply because insertion and deletion operations in an heap take logarithmic time whereas without heap they take linear time and hence are slower. The time required by Kruskal's algorithm is slightly better than Dijkstra's algorithm with heap. In Kruskal's algorithm the time for building the Maximum Spanning Tree has a large constant term.

Case where a vertex is adjacent to 20% of the vertices: This is a very large graph compared to the first case. Though it has the same number of edges, it has almost 2500000 edges. The time complexity of Dijkstra without Heap is unrelated with m [O (n_2)] but it still takes a larger time compared to the time taken in case of the Sparse Graphs.

The performance of Dijkstra with heap is similar (slightly better) to the performance of Dijkstra without heap. This happens because as m increases then O (n_2) is not necessarily very large compared to O ((n + m) log n). Both the algorithms take more time in case of a dense graph compared to the time taken in case of a sparse graph. Kruskal's algorithm takes large time in case of dense graphs compared to the time taken in case of sparse graph. This happens because the time complexity for Kruskal's algorithm is dependent on the number of edges in the graph. For Kruskal's Algorithm it takes higher time to build a Maximum Spanning Tree but once a Maximum Spanning
Tree is constructed then finding a path can be done in linear time.

Major: Industrial Engineering (ISEN)

<u>References Used:</u>

- The course book (An Introduction to Algorithms)
- Online Resources (For better understanding the python defined methods and libraries)
- CSCE 629 - Course Material & Lectures
- STAT 624 - Course Material & Lectures