ECE345 Assignment Writeup

Ryan Do (1001254117)

Munef Agag (1000973707)

March 16, 2019

*Implementation details: (a) how are you representing the graph, using an adjacency matrix or an adjacency list, and why? (b) Which shortest path algorithm did you use and why?*

We chose to represent the graph as an adjacency list. This choice was made due to the following reasons:

1. The graph is likely very sparse, due to the large expected range of influence among entities. Among all entities (nodes) that exist on the graph, most are statistically likely to not be connected. As a result, while an adjacency matrix implementation requires O(V^2) space, an adjacency list only requires O(V+E) space which is significantly lower for a sparse graph.
2. For this application of finding shortest paths from a source node to all other nodes (i.e. using Djikstra’s algorithm), we simply need to know the adjacent nodes from a given node. We do not need determine whether a specific node is adjacent to a given node. As a result, the advantage of computing the presence of a specific edge in O(1) time using an adjacency matrix is irrelevant. For determining all the adjacent nodes to a given node, an adjacency list can achieve this in less than O(V) time while an adjacency matrix takes a guaranteed tight bound O(V) time.

Dijkstra’s algorithm was the shortest path algorithm of choice. Bellman Ford’s capability to handle negative weights is not needed here since the weights (time) in the graph are constrained to be greater than 0. Dijkstra’s algorithm is also the fastest shortest-path-searching algorithm for a directed graph with non-negative weights. Its O((E+V)logV) time complexity with a binary heap implementation trumps the O(VE) time complexity of Bellman Ford. It can be improved further to O(E+VlogV) using a fibonnaci heap implementation.



In this run-time experiment, we fixed the number of nodes to be 100 and varied the number of edges from 200 to 1000 in increments of 25. As a result, we have data points for running the top 1/2 influencer computation on 33 different graphs with densities ranging from 2 to 10 evenly spaced out with a fixed number of nodes (100). It is important to note that when constructing these graphs, edges were sequentially and **randomly** picked from a superset (facebook\_large.txt) while ensuring that the final graph achieves the target node and edge counts.

In the above plot, we can observe more or less a *linear* dependence of run-time on graph density. This makes sense because Dijkstra’s O((E+V)logV) run-time converted to O(density) is O((density+1)logV) where V=100 is fixed. Something else worth noting is that the run-time variance increases as graph density increases:

Every edge in the graph is a random variable thus the distributions of the edges can vary even for a given density. For a given density, graphs with dense clusters of edges (best case only one node influencing all other nodes) could potentially run Dijkstra’s algorithm more quickly than graphs with a uniform spatial distribution of edges (edges spread evenly across all nodes). This would make sense because there would be only one topological level for a graph with just one influencer (best case).

As a result, the run-time is a random variable with variance

As density increases with a fixed number of nodes, the number of edges increases (each with the same variance) thus as per the above relationship, the variance of runtime increases.