Dayoung Lee
DL29923

EE 360C Programming Assignment 2 Report

(a) Explain the tradeoff between using adjacency matrix representation of graph and adjacency list representation of graph. Give a Big-O analysis of the memory space efficiency of your graph representation in terms of nodes N and edges M.

For adjacency list representation, we will have N adjacency lists. For an undirected graph, there are 2*M edges represented in the lists. Therefore, we have a total of O(N+2*M), or O(N+M) memory used.

I am using an adjacency matrix for my graph representation. It uses a NxN matrix, which uses $O(N^2)$ memory.

If we have a sparse graph, it would be better to use an adjacency list because it uses less memory. If we want to know if an edge exists between two nodes, *u* and *v*, we might have to traverse *u*'s entire list. If we have a dense graph, it would be better to use an adjacency matrix because it uses about the same amount of memory compared to adjacency lists, and finding whether an edge exists is faster because we just check A[u][v], which takes O(1).

(b) Give a Big-O analysis of the runtime complexity and memory complexity of the algorithm you used to find the time optimal route.

The time array is of size |V|, or N
The queue is of size O(|E|), or O(M)
So the memory complexity of my time optimal algorithm is O(|V| + |E|), or O(N + M)

Initializing the time array takes O(|V|) time, or O(N)
The while loop runs O(|V|) times, or O(N)
Removing the node with the minimum time in the queue takes O(|V|) time, or O(N)
The for loop runs a total of O(|E|) times, or O(M)
So the time complexity of my time optimal algorithm is $O(|V| + |V|*|V| + |E|) = O(|V|^2 + |E|)$, or $O(N^2 + M)$

(c) Write pseudo-code for the algorithm you have implemented to find the capacity optimal route and prove the correctness of your algorithm.

We have *cap*, which is a list containing the largest capacity to each port. Set all the capacities to the smallest value, 0. Set the capacity of *sourcePort* all to largest value, ∞. We have a queue *Q* that holds the nodes to explore. Add all node to *Q*. While *Q* isn't empty, node *u* = node with highest traversed capacity. For each node *v* that *u* is connected to, check if the capacity at *u* or

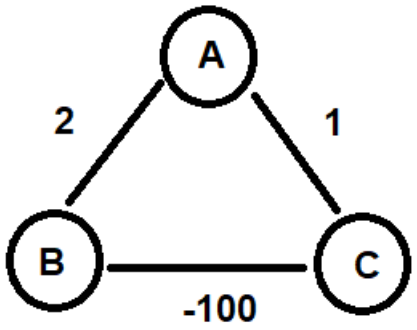
the capacity from *u* to *v,* whichever's smaller, is greater than the capacity at *v*. If it is, set the capacity at *v* to the smaller value and add node *v* to *Q*. When the while loop is finished, return the capacity at the *destPort*.

The capacity optimal route algorithm terminates with the largest bottleneck capacity for a given start port and end port.

Proof by Induction:
Base case: |S| = 1
Inductive hypothesis: Assume true for |S| = k ≥ 1
Let v be the next node added to S and let u-v be the chosen edge. The capacity of path s-v, or c(v), is the largest capacity path s-u or the capacity of edge u-v, whichever's smaller. Consider any s-v path P. Let x-y be the first edge in P that leaves S. The capacity of path P is no bigger than c(v) because P has not been traversed yet, which means that the path has a smaller bottleneck capacity. Therefore c(v) is the largest bottleneck capacity at v. When the algorithm terminates, the capacity at the end port will be the largest bottleneck capacity from the start port to end port.

(d) Is Dijkstra's algorithm able to solve graph problem with negative weighted edges? Explain why?

Dijkstra's algorithm cannot solve all graph problems with negative weighted edges. We want to find the shortest path from A to C in the graph below:



Djikstra's algorithm (based on the pseudo code given) will not find the path A-B-C as the shortest path because when it checks for B's neighbors, C is no longer in Q because C was polled from Q before B was, so the for loop will not run, and the shortest path to C will still be A-C. In other words, Dijkstra's algorithm depends on the idea that the distance is always increasing and never decreasing, so the graph cannot have negative weights.