Dhruv Sandesara
EID: djs3967
Algorithms EE360C
Programming Assignment #2 Lab Report

a) There are trade offs between using the adjacency matrix and adjacency list. The list is more efficient in the cases when the input edges are not that many. There will be a lot of memory space savings for using the list as we will not have to save any of the connections that are not present. On the other hand, while using an Adjacency matrix we will have to set the edge weights for non existent edges to 0. This will lead to huge memory wastes as we will be storing unnecessary information. But when we look at the cases for a large input of edges, we start seeing computational benefits of using the Matrix. This is as the access of checking whether an edge exists between two points is O(1) whereas we will have to iterate though the entire list to just check if an edge exists between two nodes. Considering the Memory space efficiency the Matrix has O(n^2) while for the list has O(n+2m) as it is undirected edges. We have the n for each node and each m is the edge and 2m as we will have to store it both the ways.
b) As I used the Adjacency Matrix, my Memory complexity is O(n^2). My runtime complexity is O(n+n + n(n)+m) where the first 2 n are the setting the infinity distance and populating the queue. Then next n term is because of the for loop. Inside the for loop we have the n for finding the minimum. We are also adding all of the neighbors and changing their distance values. But the most amount of time we can do this is for all the m edges which is why we ca take it out. Thus my final time complexity is **O(n^2 +m)= O(n^2).** The reason for this bottle neck is finding the minimum value in my queue. As I am implementing the queue with an ArrayList it takes n^2. If I use an priority-queue which uses a binary heap in the background and can figure out the the minimum in Log n my complexity could be reduced by nLogn +m.
c)

```
1: Function Dijkstra(Graph, source, sink)
2: for each vertex v in Graph // Initialization do
3: cap[v] := 0 ; // Unknown capacity function from source to v
5: end for
6: cap[source] := Integer.MAXValue ; // Capacity from source to source
7: Q := the set of all nodes in Graph ; // All nodes in the graph are unoptimized - thus are in Q
8: while Q is not empty: // The main loop
9: u := vertex in Q with highest capacity in cap[] ;
10: if cap[u] = 0 then
11: break;
12: end if
13: remove u from Q ;
14: if u = sink then
15: break;
16: end if
17: for each neighbor v of u: // where v has not yet been removed from Q. do
18: alt := min(cap[u], cap between (u,v));
19: if alt > cap[v]: // Relax (u,v,a) then
20: cap[v] := alt ;
23: end if
24: end for
25: end while;
26: return cap[sink] ;
27: end Dijkstra
```

The proof that my algorithm works to get the max capacity:
Proof By induction.
For Size one we know the capacity to the port itself will be infinity. So it is the largest capacity.
Inductive hypothesis: We assume that there are n vertex where each vertex has the correct largest capacity to that vertex. Then we add the last vertex and try to find the largest capacity to that vertex. The extra information provided to us is this new node's neighbors and the capacity to each of the new routes.
Now according to our algorithm we will try out all the routes to this new node and figure out the the Largest capacity to reach this node. Then it will assign this found capacity as the optimally large capacity to this route. But our algorithm does not end here. We want to now recompute the neighbors of the new nodes capacity as well, by making them chose the minimum of new node capacity and the edge capacity. Then we compare this temp capacity to their previous capacity and keep the largest one. This way we have anticipated all possibilities of capacities changing and maintaining an updated max capacity for each node. Hence our algorithm works.
Hence Proved.

d) No the Dijkstra's algorithm will not be able to solve graph with negative weighted edges. This is as Dijkstra's works on the fact that as all the weights are positive, and adding any edge will make the path longer. Thus Dijkstra's can reliably remove the node from the open set. On the other hand, If we have negative edge weights, there will be a chance when adding an extra node, will reduce the distance to that node. This is the case when Dijkstra's will break down because of negative nodes. Thus It does not work. For example A-B is 5, A-C is 3 and B-C is -2. The start is A so A is 0, B is inf., and C is inf. After checking and removing a B is 5, and C is 2. As C is now the min, it will just return it. It never gets to check the path A->B-> C as it assumes that it has more nodes will just ad to the distance between A-B.