Daniel Diamont
UT EID: dd28977
Class: EE360C
Section: 15775 (Dr. Julien)

Lab 2 Report (Dijkstra's Algorithm)

(a) Explain the trade off 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.

Let m be the number of edges in the graph, and let n be the number of nodes in the graph.

The trade off between using an adjacency matrix representation and using an adjacency list is that while an adjacency matrix allows edge lookup in O(1) time versus an adjacency list's O(m) time, in the worst case, iteration over the entire matrix will always take $O(n^2)$ time. In comparison, iteration over the entire adjacency list will be $O(n+m)$ in the amortized case (sparse graph), but will also be $O(n^2)$ in the worst case (dense graph). Additionally, an adjacency matrix will always take up $\Theta(n^2)$ space, while for a sparse graph the adjacency list offers the advantage that it will take up $\Theta(n+m)$ space.

For this lab, to run Dijkstra's algorithm, I chose to represent the graph with an adjacency lookup, since I did not require fast edge lookup functionalities, and I benefit from taking $\Theta(n+m)$ memory space instead of $\Theta(n^2)$ space.

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

Let m be the number of edges in the graph, and let n be the number of nodes in the graph.

To find the time optimal route, I used Dijkstra's algorithm implemented with a priority queue in the form of a min-heap. The initialization of a heap can be accomplished in $O(n)$ time. Additionally, a min-heap allows extractions of the minimum key as well as changing the key of an element within the heap in $O(logn)$ time. Since at most there key of a node can change *m* times, then there can occur at most $O(m)$ change-key operations for a total of $O(mlogn)$ time for all the priority queue operations. Additionally, the time to minimize the quantity $d'(v)=min_{(edge(u,v))}[d(u)+w(u,v)]$ for each node over the span of the entire program will take $O(m)$. Thus, the total upper bound on the runtime can be represented as: $O(n)+O(m)+O(mlogn)=O(mlogn)$.

In the best case where we have a sparse graph, $m=\Theta(n)$, we can maintain the runtime complexity of $O(nlogn)$. In the worst case where we have a very dense graph, $m=\Theta(n^2)$, then our runtime complexity degrades to $O(n^2logn)$. It is in this special case that a linked list implementation of a priority queue would yield a runtime of $O(n^2)$. Thus, if I knew beforehand that a graph will be extremely dense, I would choose the linked list implementation in favor of the min-heap priority queue.

As far as memory complexity goes, my implementation of Dijkstra's algorithm will take $\Theta(n+m)$ to store the graph in the form of adjacency lists. The size of the queue can be at most the number of nodes in the graph -- $\Theta(n)$. Thus, the space complexity is $\Theta(n+m)$.

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

```
1: Function findCapacityOptimalPath(Graph, source,sink )
2: for each vertex v in Graph // Initialization do
3:
cap[v ] := negative_infinity ; // Unknown capacity function from source to v
4:
previous[v ] := undefined ; // Previous node in optimal path from source
5: end for
6: cap[source] := infinity ; // Distance 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 largest capacity in cap[] ;
10: if cap[u] = negative_infinity 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:
weight = capacity between (u,v)
19:
if cap[u] < weight: // Relax (u,v,a) then
20:
if(cap[v] < cap[u]:
21:
then cap[v]: = cap[u]
22:
endif
23:
else: // cap[u] >= weight
24:
if cap[v] < weight
25:
then cap[v]: = weight
26:
end if
27:
end if
28: predecessor[v]:= u
29: decrease-key v in Q; // Reorder v in the Queue
30: end for
31: end while;
```

32: return dist[sink] ;
33: end findCapacityOptimalPath

Proof of Termination:

The loop invariant is that at the end of each iteration of the for loop, the priority queue contains all of the nodes that we do not have the largest bottleneck capacity. Thus, at the beginning, all of the nodes in the graph are also in the queue. Since the algorithm will iterate over all of the nodes in the graph, the loop can occur at most *n* times, after which the algorithm will terminate because we will have found the maximum bottleneck capacity for each node in the graph.

Proof of Correctness:

Using the proof of termination, we know that at the beginning of the algorithm the priority queue Q – implemented via max-heap – will contain all of the nodes in our graph S except the source node s, since we do not know the maximum capacity for any of the nodes except that of the source. Since we are at the source, we know that in theory we can send an unlimited number of ships to itself because if the source is the destination, the ships do not actually have to travel anywhere, so there is no capacity limit. In other words, cap[s] = oo. At the end of the first iteration, s is removed from the priority queue and placed in the set S where we know the maximum bottleneck capacity of each node.

Suppose this holds for |S| = k nodes where $k \geq 1$. Then, all of the nodes in S are nodes for which we know their maximum bottleneck capacity on a path from source node s to each of the nodes. In iteration k+1 of the loop, let us grow the set S to size |S| = k + 1 by adding a destination node v, such that the edge (u,v) is the final edge on our s-v path. Thus, by the inductive hypothesis, we will know the maximum possible bottleneck capacity for node v. Let the s-v path be called P.

Now, consider any other path P' such that it contains a different path to s-v. We wish to show that the maximum bottleneck capacity of path P' is at most that of our original path P. Let y be a node in the path P' and in the set S, such that the edge (y,v) leaves S and connects to the edge v in the $k+1^{th}$ iteration where at the beginning $v \notin S$. Once again, because of the loop invariant, we know that at the end of the iteration, v will have been removed from the priority queue and will be placed in S because the algorithm will have chosen the edge which maximizes the bottleneck capacity out of all the possible edges. This maximization expression is given by:

$cap[v] = max_{(e=(u,v)\in Adj(v))}[min(cap[u], w(u,v))]$ where w is the weight function that represents the capacity of each edge.

In our case where we are considering path P and P' the maximization expression can be denoted as:

$cap[v] = max[min(cap[u], w(u,v)), min(cap[y], w(u,v))]$.

In other words, in the $k+1^{th}$ iteration would have considered both the edge (y,v) and the edge (u,v) and would have chosen the one to maximize the bottleneck capacity of node v. Since v was added via edge (u,v) instead of (y,v), then we can conclude that the bottleneck capacity given by taking edge (y,v) in path P' is less than or equal the bottleneck capacity given by taking edge (u,v) in path P.

Thus, at the end of the algorithm, we can conclude that there will not be any nodes left in the priority queue, and that all nodes will be in the set S where we will now the maximum capacity bottleneck of each node.

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

No. In Dijkstra's algorithm, once a vertex is removed from the priority queue after a relax step, the algorithm assumes that it has found the shortest path to that vertex, and will visit that vertex again during the course of the algorithm. With negative weights, it is possible that Dijkstra does not find the shortest path.

Consider the case where there exist three nodes x,y,z with edges (x,y), (y,z), (x,z) to form a triangle. Now consider that the weights of each edge are as follow:

w(x,y) = 10
w(y,z) = -20
w(x,z) = 1

Now, running Dijkstra's algorithm from source vertex x will first relax node z. It will remove z from the priority queue and assume that the shortest path is that of edge (x,z) with a weight of 1. The algorithm will fail to find the path x→ y → z – which would yield a shortest path of -10 – because z will not be in the priority queue by the time the algorithm visits vertex y.