Data Structures and Algorithms Homework 6

Due Wednesday Oct 9; Joseph Sepich (jps6444)

1 Problem 1 Dijkstra's Algorithm

Collaborators: None

1.1 Part a

1. s
2. a
3. c
4. d
5. b

1.2 Part b

The edge needs to be 3 or greater, since the vertex removed after a is c, and if this edge is 2 or less it could be removed next (would be tied or less than c at the step after vertex a). The edge also needs to be 5 or less so it will be removed before the b vertex, whose distance is 7 when a is removed.

If tied vertices are removed in alphabetical order then the constraint on a to d is [2, 5], otherwise the strict bounds are [3, 5].

2 Problem 2 Negative Weights

Recall our inductive proof on the correctness of Dijkstra's algorithm. We know that the first iteration out of the source node to the connected vertices will have the correct path. This is the base case, and s and its smallest connected edge (even negative) is added to the set S, which denotes all vertices with known shortest path. If we search then from this smallest path s -> v, where v is the furthest vertex in the set S, it is guranteed to be the shortest path. Adding u, a vertex outside the set v, we compared it's current distance with the distance of v plus the length to u. We only add the vertex to the path if the distance of v plus the length of the edge to u is shorter than its current path distance. Since the distance array is intitalized with the only negative edges, the length value used in all of these comparisons will only be strictly positive, **so the vertex u added cannot have an edge coming from a vertex not in S that has a negative length to decrease distance, the shorest path vertex to u (not in S), must come from the set S**. This is what had broken the algorithm with negative edges before, but since all the negative edges in in the array at the start, Dijkstra's algorithm will succeed in the graph.

3 Problem 3

An efficient algorithm would be to run Dijkstra's algorithm twice to start. The first run you use s as the source node, and the second run you use t as the source node. You know have data on the shorest past from city s to every city and from city t to every city. This also means you know that base time/length from s to t. You can then loop through the edges in E' and for each edge from vertex u to v, you can sum the distance from s to u, the length of the edge e', and the distance from v to t to get the length of that possible new path. For each of these paths store the value, but if you find a path of a lesser value store that (starting with this value assigned to the current value of the path from s to t). Then you can just return this distance and edge e' as the new edge to add. This algorithm is efficient, because it runs through Dijkstra twice, then just loops through the edges in E'. This has a running time of O((|V| + |E|)log|V| + |E'|).

4 Problem 4

All you have to do to create this modified version of Bellman-ford is instead of running update on all the edges |V| - 1 times, you run the update function at most k times. If after these k updates of all the edges the value of the distance to t is still infinity, then you know you cannot make it there in k edges, otherwise you know you have the shorest path containing at most k edges. This has to do with the way Bellman-Ford finds shortest paths. Since everything is initialized to infinity, but the source node, the shortest paths to edges that are not s are found in a BFS type manner, where all vertices 1 edge away from s are update in the first iteration, then the second iteration updates everything within 2 edges of s, and since update by defintion will only make a path smaller, this algorithm will find the shortest path containing k edges.

5 Problem 5

5.1 Part a

Given the constraint that every conversion rate from i to j must be greater than or equal to zero, you can use Bellman-Ford algorithm on a graph where you transform all the edges by multiplying by negative one. These set of edges would be the minimum of the negation, but the maximum of the original set of edges. The max path in this case would be the most advantageous sequence of conversions, because it would provide us with the maximum possible mileage for the destination airline from the source airline. As we know Bellman-Ford runs with time complexity of O(|V|*|E|), because it updates every edges |V| - 1 times.

1. Negate edges
2. Run Bellman-Ford
3. Shortest Path in negated edges is max path/desired path in non-negated edges.

5.2 Part b

Since we are using negative edges and the Bellman-Ford algorithm in part a, checking for accretion is the same as checking for a negative cycle. If you negate all the edges and run Bellman-Ford |V| - 1 times, then you found the min path of the negated eges (max path for the original graph). If you update all the edges one more time, and any edge actually **does** update, then you know that your graph with negated edges contains a negative cycle, which implies you can always obtain a smaller and smaller path. Since these edges are negated you know that this same negative cycle would be a profitable cycle in the original graph that would always obtain a larger and larger path. Therefore to determine if accretion exists you just need to run the previous algorithm in part(a), but add an extra round of updates to check for a negative cycle. Since we run update on |E| edges |V| times, the running time is still O(|V|*|E|).