CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Homework #5

December 5, 2019

Name: FIRST LAST *Extension: Yes/No*

Problem 1. Describe and analyze an algorithm that finds the *second smallest spanning tree* of a given undirected connected (weighted) graph G, that is, the spanning tree of G with smallest total weight except for the minimum spanning tree. You may assume that all edge weights are *distinct*s in the graph (recall that in this case the minimum spanning tree would be unique). **(20 points)**

Hint: First prove that the second smallest spanning tree of G is different from the MST of G in only one edge. Then use this to design your algorithm by starting from an MST of G and checking which edge should be changed to obtain the second smallest spanning tree.

Solution.

Algorithm:

- We use Kruskal's algorithm to find the MST
- Now, for each edge in MST, we remove the edge that is in MST and calculate another spanning tree
- We will run the Kruskal's algorithm on the new tree that we found and compare the cost with the last tree
- Of course we will be keeping track of the smallest spanning tree
- In the end we will add the edge back to tree and return the smallest MST

Proof of correctness:

- We will start by running the Kruskal's algorithm on the given undirected graph which will give us the (first) MST
- Once we have that, we will reomve one edge from the MST that we got and re run the Kruskal's algorithm on the new tree, which is different by one edge from the previous tree
- We will keep track of the spanning trees we get and check which one is the smallest
- In the end we will add the edge back and return the smallest spanning tree

Time Complexity:

- Kruskal's algorithm takes O(m log m)

- Since we repeat Kruskal's algorithm inside of a loop after removing the edge, it will now take O(m^2 log m)

Problem 2. You are given a weighted graph $G(V, E)$ (directed or undirected) with positive weight $w_e > 0$ on each edge $e \in E$ and two designated vertices $s, t \in V$. The goal in this problem is to find a s-t path P where the maximum weight edge of P is minimized. In other words, we define the weight of a s-t path P to be $w_{MAX}(P) = \max_{e \in P} w_e$ and our goal is to find a s-t path P with minimum $w_{MAX}(P)$ (recall that in the shortest path problem, we define weight of a path to be sum (instead of max) of the weights of edges).

Design and analyze an $O(n + m \log m)$ time algorithm for finding such a path in a given graph. **(20 points)**

Hint: In this rare instance, it may be easier to modify Dijkstra's algorithm directly, instead of doing a reduction (although the latter option is also completely possible) – just remember to prove the correctness of your modified algorithm from scratch. Another option would be to instead use the DFS algorithm combined with a clever binary search approach.

Solution.

Algorithm:

- We will basically use the modified version of Dijkstra's algorithm

- Unlike Dijkstra's algorithm we will consider maximum weight edge from source rather than distance from source

- Now for every neighbour or adjacent vertex, we will check if it's in min heap, and if so, we will check if the weight value of that vertex is more than the weight of u-v edge and u itself, where u is the starting and v is the adjacent vertex, then we will update the weight of v to the maximum of either the weight of u or the weight of the edge from the adjacent vertex to u

- We will repeat this process for all vertices until sink or end vertex is reached

Proof of correctness:

- First step is to creating a min heap containing min heap and maximum edge weight from source. We will initialize it with distance infinty and source distance 0

- As mentioned in the algorithm, this is a modified version of Dijkstra's algorithm

- We will, for some adjacent vertex v, update the min heap to the maximum of either the weight of u or the weight of the edge from the adjacent vertex to u, if the weight value of that vertex is more than the weight of u-v edge and u itself

- We will be repeating the process for every vertex

- This way, we can minimize the maximum weight of edge in finding path from source to destination.

Time Complexity:

- This is basically a modified version of Dijkstra's algorithm, so the time complexity will be $O(n+m\log m)$

Problem 3. You are given a directed graph $G(V, E)$ with two designated vertices $s, t \in V$, and some of the vertices in G are colored *blue*. Design and analyze an $O(n + m \log m)$ time algorithm that finds a path from s to t that uses the *minimum* number of blue vertices (the path can use any number of non-blue vertices).

(15 points)

Hint: Unlike the previous problem, the easiest solution here is a simple graph reduction.

Solution.

Algorithm:

- For the graph, we can mark the length of some edge (s,t) in graph G, 1 if t is a blue node and 0 if otherwise

- Then we can run Dijkstra's algorithm that will give us the path with minimum blue vertices

Proof of correctness:

- We will start by marking the edges 0 or 1

- For some edge (s,t), we will mark the length of the edge to be 1 if t is a blue node and 0 if it is not a blue node

- We know from Dijkstra's algorithm that it select the least weighed path, and thus, we will be igonoring the paths that does contain the blue node because of the weight of other vertices are lesser than the blue vertices

- This will give us a path from s to t with least number of blue vertices

Time Complexity:

- We basically run Dijkstra's algorithm to get the path containing least number of blue vertices, which takes $O(n + m \log m)$ time

---

Problem 4. In the class, we studied the single-source shortest path problem and saw Bellman-Ford and Dijkstra's algorithm for solving this problem. A related problem is the *all-pairs* shortest problem where the goal is to, given a directed weighted graph, compute the shortest path distances from every vertex to every other vertex, i.e., output an $n \times n$ matrix D where $D_{ij} = dist(v_i, v_j)$, namely, the weight of the shortest path from v_i to v_j. Our goal in this question is to design an $O(n^3)$ time algorithm for this problem.

(a) Design an $O(n^2)$ time algorithm that takes as input a weighted directed graph G and any arbitrary vertex v in G, and constructs a new directed graph $G'(V', E')$ with weighted edges such that $V' = V \setminus \{v\}$, and the shortest path distances between any two vertices in G' is equal to the shortest path distances between them in G. **(10 points)**

Solution. Solution to Part (a) goes here.

(b) Now *assume* we have already computed all-pairs shortest path distances in G'. Describe an $O(n^2)$ time algorithm to compute the shortest-path distances in the original graph G from v (the vertex chosen in part (a)) to every other vertex in V, and from every other vertex in V to v. **(10 points)**

Solution. Solution to Part (b) goes here.

(c) Combine your solutions for parts (a) and (b) to design a *recursive* all-pairs shortest path algorithm that runs in $O(n^3)$ time. **(5 points)**

Solution. Solution to Part (c) goes here.

Problem 5. You are given an *undirected* graph $G(V, E)$, with three vertices u, v, and w. Design an $O(m+n)$ time algorithm to determine whether or not G contains a (simple) *path* from u to w that passes through v. You do *not* need to return the path. Note that by definition of a path, no vertex can be visited more than once in the path so it is *not* enough to check whether u has a path to v and v has a path to w. **(20 points)**

Hint: While it may not look like it, this is a maximum flow problem. Create a flow network from G by making all edges in G bidirected and adding a new source vertex s that is connected to u and w. Assign an appropriate capacity to the *vertices* in this graph, and identify an appropriate target vertex t. Finally, find a maximum flow using the Ford-Fulkerson algorithm and argue that this only takes $O(m + n)$ time.

Solution.

Challenge Yourself. Consider Problem 2 again and suppose now that the graph is *undirected*. Design and analyze an algorithm for the same problem with the improved running time of $O(n + m)$. You may use the fact that there is an algorithm that given an array of size k, can find the *median* of the array in $O(k)$ time.