Homework #6 Solutions
Introduction to Algorithms
601.433/633
Spring 2020

Due on: Tuesday, April 21st, 12pm

1 Problem 1 (15 points)

Let $G = (V, E)$ be an undirected graph with *distinct non-negative* edge weights. Consider a problem similar to the single-source shortest paths problem, but where we define path cost differently. We will define the cost of a simple s-t path $P_{s,t} = \{e_1, e_2, \ldots, e_k\}$ to be

$$c(P_{s,t}) = \max_{e \in P} w_e$$

The cost of a path is now just the largest weight on that path, rather than the sum of the weights on the path. Give an algorithm that takes as input an undirected graph $G = (V, E)$ with *non-negative edge weights*, and a vertex $s \in V$, and computes the path from s to every other node in G with the least cost under cost function $c(\cdot)$. That is, for each $t \in V$, find a simple path $P_{s,t} = \{(s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, t)\}$ connecting s to t, such that, letting S denote the set of all simple paths connecting s and t, $c(P_{s,t}) = \min_{P \in S} c(P)$.

Your algorithm should run in $O(|E| + |V| \log |V|)$ time. Prove the correctness of your algorithm and its runtime. Hint: note the similarities between Dijkstra's algorithm and Prim's algorithm.

Proof. It suffices to change the RELAX function in Dijkstras algorithm. Nothing else in the analysis changes our invariant is still that the distance stored for each vertex is always the cost of the best path to that vertex discovered so far.

Specifically, we will change the way we update the $v.d$ field for each vertex v (recall that $v.d$ tracks what we currently believe to be the length of the shortest path to vertex v). Suppose we are currently at vertex u, and $v \in \text{Adj}(u)$. If $\max\{u.d, w(u,v)\} < v.d$, then we will set $v.d = \max\{u.d, w(u,v)\}$. The correctness of our new version of Dijkstras algorithm follows from the correctness of Dijkstras algorithm and the fact that our new version of the RELAX function still obeys the invariant used in Dijkstras, namely that the $v.d$ field is always the cost of the lowest-cost path from the source node to v explored so far.

CLaim: The relaxation algorithm maintains the invariant that $v.d \geq d(s,v)$ for all $v \in V$.

We will induct on the number of steps of the algorithm. Consider RELAX(u,v,w). Let $u.d \geq d(s,u)$ by the induction hypothesis. By the triangle inequality, $d(s,v) \leq \max\{d(s,u), d(u,v)\}$. This means that $d(s,v) \leq \max\{u.d, w(u,v)\}$, since $u.d \geq d(s,u)$ and $w(u,v) \geq d(u,v)$. So setting $v.d = \max\{u.d, w(u,v)\}$ is safe.

□

2 Problem 2 (15 points)

You are given a simple (no self-edges or multi-edges) weighted graph $G = (V, E)$ and its minimum spanning tree T_{mst}. Somebody added a new edge e to the graph G. Let's call new graph $G' = (V, E \cup e)$. Devise an algorithm which checks if T_{mst} is also a minimum spanning tree for the graph G' or not. Your algorithm should work in $O(|V|)$ time. Prove correctness of your algorithm and provide running time analysis.

You can assume that all edge weights in G' are distinct. T_{mst} and G' are given to you as adjacency lists.

Proof. **Algorithm.** Suppose the new edge is $e(u,v)$, use DFS starting from node u to find the path from node u to v in T_{mst}. Then find the maximum weight w_{max} in this path $u \rightsquigarrow v$. If the w_{max} is less than the weight of new edge e, it indicates that T_{mst} is also a MST of the graph G'. Otherwise, G' has different MST with G.

Correctness. If we add the new edge $e(u,v)$ to the T_{mst}, there must exist a cycle which contains this new edge e in this new graph. Since there is a path from node $u \rightsquigarrow v$ in the T_{mst}, adding this new edge $e(u, v)$ could be considered as another path from $u \rightsquigarrow v$. Thus the path $u \rightsquigarrow v$ and e would form a cycle C. This cycle obviously belongs to G'. If e has maximum weight among all edges in C, then according to result from the first problem we can conclude, that e doesn't belong to the MST of G', thus T_{mst} is still an MST for G'. If e doesn't have a maximum weight, then some other edge $e' \in C$ does, while all other edges are from T_{mst}. Thus we found an edge e' which can't be in MST of G' according to the result from problem 1, but it does belong to T_{mst}, therefore we can conclude that T_{mst} isn't an MST for G'.

Running Time. The number of edge in MST equals to $|V| - 1$. Since the graph was stored in adjacency list, the running time of DFS is $O(|V| + |E| - 1) = O(2|V| - 1) = O(|V|)$. It will cost $O(|V|)$ to find the path $u \rightsquigarrow v$ in T_{mst}, and $O(|V|)$ to find the maximum weight in this path $u \rightsquigarrow v$. Comparing the maximum weight with the weight of new edge will take constant time. Therefore the total running time of algorithm is $O(|V|)$. $\square$

3 Problem 3 (20 points)

An airline has n flights. In order to avoid "re-accommodation", a passenger must satisfy several requirements. Each requirement is of the form "you must take at least k_i flights from set F_i". The problem is to determine whether or not a given passenger will experience "re-accommodation". The hard part is that any given flight cannot be used towards satisfying multiple requirements. For example, if one requirement states that you must take at least two flights from $\{A, B, C\}$, and a second requirement states that you must take at least two flights from $\{C, D, E\}$, then a passenger who had taken just $\{B, C, D\}$ would not yet be able to avoid "re-accommodation".

Given a list of requirements $r_1, r_2, \ldots, r_m$ (where each requirement r_i is of the form: "you must take at least k_i flights from set F_i"), and given a list L of flights taken by some passenger, determine if that passenger will experience "re-accommodation". Your algorithm should run in $O(L^2 m)$ time.

Hint: You should just need to show how this can be reduced to a network flow problem and use a blackbox algorithm solving the flow problem. Prove that your reduction is correct and that the runtime of the algorithm you use will be $O(L^2 m)$

on the reduction you give.

Proof. **Algorithm.** First, we can create a bipartite graph where on the left we have one node for each requirement r_i, and on the right we have one node for each flight taken by the passenger. Put an edge between requirement r_i and flight j if $j \in F_i$. give all these edges a capacity of 1. Now let's create a source with an edge of capacity k_i to each node r_i on the left, and create a sink with an edge of capacity 1 to it from each flight node.

Claim. The passenger can avoid "re-accommodation" if and only if there is a flow of value $k = \sum_i k_i$.

First, if the passenger can avoid being "re-accommodated" , then there must be a flow of value k, since we flow k_i units into each node r_i, split this flow into k_i pieces of size 1 going to the k_i flights used to satisfy the requirement, and finally send all that flow to the sink. At most 1 unit of flow will be on any edge into the sink because we are guaranteed that no flight is being used to satisfy more than one requirement. In the other direction, if there is a max flow of value k, then there must be one in which all flows are integral (by the Integral Flow Theorem). Such a flow must therefore, for each requirement r_i, have k_i edges exiting it with flow equal to 1. In addition, no two flow-carrying edges from different requirements go to the same flight node. Therefore, these edges with flow 1 indicate a legal way to satisfy all the requirements. $\square$