CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2021**

Homework #4 Solutions

April 9, 2021

Problem 1. You are given the map of n cities with m bidirectional roads between different cities. You are asked to construct airports in some of the cities such that each city either has an airport itself or there is a way to go from this city to a city with an airport using the given roads—moreover, you can also construct a road between any two cities if there is no road between them already. Finally, you are told that the cost of constructing an airport is a and the cost of connecting any two cities by a new road is r.

Design and analyze an $O(n+m)$ time algorithm that given the number n of the cities, the m roads between them, and the costs a and r, outputs the locations of airports and roads to be constructed to satisfy the conditions above, while having the minimum possible cost. **(25 points)**

Examples:

1. *Input:* $n = 4$ cities with $m = 2$ roads $(1,2),(2,3)$, cost of constructing an airport $a = 7$, and constructing a road is $r = 5$.

   Output: The minimum cost needed is 12 – we construct an airport in city 4 and connect it this city via a road to any of the cities 1, 2, or 3 (chosen arbitrarily).

2. *Input:* $n = 4$ cities with $m = 2$ roads $(1,2),(2,3)$, cost of constructing an airport $a = 7$, and constructing a road is $r = 8$.

   Output: The minimum cost needed is 14 – we construct an airport in city 4 and another one in any one of the cities 1, 2, or 3 (chosen arbitrarily).

Solution. *Algorithm.* We create a graph with a vertex for every city and an edge between two cities if there is a road between them. We then compute the connected components $C_1,\ldots,C_k$ of this graph (for $k \geq 1$) using DFS or BFS. If $a \leq r$, i.e., cost of airport is less than road, we build k airports, each in one arbitrary city of these connected components. Otherwise, if $a > r$, i.e., cost of airport is more than road, we build one airport in some arbitrary city of connected component C_1, and then build $k-1$ roads connecting this city with airport to one arbitrary city of each remaining connected component.

Proof of correctness. We first claim our algorithm is *feasible*, that is, in the output, every city is connected to an airport. When $a \leq r$, every connected component has an airport and thus cities in that components are connected to the airport. When $a > r$, by adding the new roads, the entire graph is one connected component and since one of the cities now has an airport, every city can reach an airport.

We now prove *optimality*, that is, the cost of our solution is minimized. Consider any solution which builds ℓ airports. In order for this solution to be feasible, the final graph, after building new roads, must have at most ℓ connected components, or else some component would be without an airport. If the final graph has at most ℓ components, then at least $k-\ell$ new roads were created, since creating a road decreases the number of components by at most 1. Thus any solution which builds ℓ airports must cost at least $(k-\ell)\cdot r + k\cdot a = k\cdot r + \ell\cdot(a-r)$.

When $a \leq r$, our algorithm gives a solution of cost $k \cdot a$, which is optimal given the previous lower bound because $k\cdot a \leq k\cdot r + \ell\cdot(a-r)$ for any choice of $1 \leq \ell \leq k$ (when $a \leq r$).

When $a > r$, our algorithm gives a solution of cost $(k-1)\cdot r + a$, which is optimal given the previous lower bound because $(k-1)\cdot r + a \leq k\cdot r + \ell\cdot(a-r)$ for any choice of $1 \leq \ell \leq k$ (when $a > r$).

Runtime analysis. The algorithm runs in time $O(n+m)$ for running DFS or BFS to find the connected components.

Note: In the class, we went over DFS for finding spanning tree of a connected graph. To make it work on a graph which is not connected and instead finds a spanning tree for each component, we can do as follows: Run DFS from a vertex to find the spanning tree of this component; remove all these vertices from the graph, and recurse. The runtime of each connected component is proportional to number of vertices and edges *inside this component* and hence this algorithm in total requires $O(n+m)$ time.

Problem 2. We say that an undirected graph $G=(V,E)$ is **2-edge-connected** if we need to remove *at least two* edges from G to make it disconnected. Prove that a graph $G=(V,E)$ is 2-edge-connected if and only if for every cut $(S, V-S)$ in G, there are *at least two cut edges*, i.e., $|\delta(S)| \geq 2$. **(25 points)**

Solution. We prove each part separately:

- If G is 2-edge-connected then every cut in G has at least 2 cut edges. We prove this by contradiction. Suppose G is 2-edge-connected but there exists a cut $(S, V-S)$ with only one cut edge e. Consider the graph $G-e$ obtained by removing the edge e from G; the cut $(S, V-S)$ now has zero cut edges in $G-e$ and as proven in Lecture 9, this means $G-e$ is disconnected. This contradicts the fact that we needed to remove two edges from G to make it disconnected.
- If G is not 2-edge-connected then there exists a cut in G with less than 2 cut edges. Since G is not 2-edge-connected, there is an edge e such that $G-e$ is disconnected; let S be any connected component of $G-e$ and note that $(S, V-S)$ is a cut in G with only one cut edge e, as desired.

Problem 3. The Muddy City consists of n houses but no proper streets; instead, the different houses are connected to each other via m bidirectional muddy roads. The newly elected mayor of the city aims to pave some of these roads to ease the travel inside the city but also does not want to spend too much money on this project, as paving each road e between houses u and v has a certain cost c_e (different across the muddy roads). The mayor thus specifies two conditions:

- Enough streets must be paved so that everyone can travel from any house to another one using only the paved roads (you may assume that this is always possible);
- The paving should cost as little as possible.

You are chosen to help the mayor in this endeavor.

(a) Design and analyze an $O(m \log m)$ time algorithm for this problem. **(10 points)**

Solution. *Algorithm:* Let us construct a weighted undirected graph $G=(V,E)$ as follows.

The vertex set V is the set of all houses in Muddy City. The edge set E is the set of all bidirectional roads in Muddy City. The weight of any edge e from u to v is the cost of paving the road from u to v.

We find the minimum spanning tree T of G with Kruskal's algorithm, and pave the edges in the tree.

Proof of Correctness: To make the Muddy city connected we need to pick at least a spanning tree of the muddy roads to turn into a street. Among all spanning trees, MST will have the minimum cost by definition, thus proving the correctness of the algorithm.

Runtime: We create the graph $G=(V,E)$ in $O(n+m)$ time and it has n vertices and m edges. The runtime of Kruskal's algorithm on this graph is $O(m \log m)$. Since $m \geq n-1$ (otherwise, we can never make the city connected), we have that the runtime is $O(m \log m)$.

(b) The mayor of a neighboring city is feeling particularly generous and has made the following offer to Muddy City: they have identified a list of $O(\log m)$ different muddy roads in the city and are willing to entirely pay the cost of paving *exactly one* of them (in exchange for calling the new street after the neighboring city).

Design and analyze an $O(m \log m)$ time algorithm that identifies the paving of which of these roads, if any, the mayor should delegate to the neighboring city to further minimize the total cost—note that if you decide to pave one of the roads payed by the neighboring city, you only need to pay a cost of 1 (for making a plaque of the name of the street). **(15 points)**

Hint: Design an algorithm that given a MST T of a graph G, and a single edge e, in only $O(m)$ time finds an MST T' for the graph G' obtained by changing the weight of the edge e to 1.

Solution. *Algorithm:* We first create the graph $G = (V, E)$ in the same way as part (a) and compute an MST T of G.

For any edge e_i that the other city is willing to pay for, define the graph G_i as the graph obtained from G by changing the cost of the edge e_i to 1.

Following the hint, run the following algorithm to find an MST T_i of graph G_i, using MST T of G:

- Add the edge $e_i = (u_i, v_i)$ to T;
- Run DFS from u_i to v_i in T to find a cycle in $T + e_i$ containing the edge e_i (recall that adding an edge to a tree always creates one cycle).
- Let f_i be the heaviest weight edge of this cycle, and return $T_i = T_i + e_i - f_i$ as an MST of G_i.

Compute the minimum weight tree among all T_i for each edge e_i, and return the minimum weight one (meaning we are asking the other city to pay for e_i and we pave the remaining roads in T_i ourself).

Proof of Correctness: Firstly, if we decide to ask the neighboring city to pay for an edge e_i, then we will end up with the same situation as part (a) with the only difference that cost of edge e_i is now 1 (we only need to pay for the plaque). By part (a), this means that *if* we decide to go with option e_i, then our cost will be equal to the weight of MST of G_i. As we are finding the minimum weight MST among all options e_i, the returned solution is correct; thus, we only need to prove that T_i is indeed an MST of G_i. To do this, we prove the following claim.

- **Claim:** For any graph H, if f is the heaviest weight of a cycle in H, then any MST of $H - f$ is also an MST for H.
- **Proof:** We prove that there is an MST T of H that does not contain f; as every MST of $H - f$ will have weight equal to T, this means all those MSTs will also be a MST for H.

  To prove this, consider a MST T' of H. If $f \notin T'$, we are already done by setting $T = T'$. Otherwise, consider the following tree; remove f from T' which partitions it into two connected components; add one of the edges of the cycle that f is its maximum weight edge to $T' - f$ to make it connected. Since $w_f \geq w'_f$, we obtain that $T - f + f'$ is also another MST; but this MST does not contain f as desired.

Using the above claim, we can argue that for each G_i, an MST of $T + e_i$ is also an MST of G_i; this is because for any edge $f \in G_i - (T + e_i)$, f is the heaviest weight edge of some cycle in G_i as we now prove: consider adding the edge f to T and look at the cycle C obtained from this. In this cycle, f has to have the heaviest weight as otherwise we can switch the heaviest weight edge with f, and getting a spanning tree with strictly smaller weight, contradicting that T was an MST of G. As a result, we only need to prove that T_i is an MST of $T + e_i$ to get that T_i is an MST of G_i as well.

Now consider the graph $T + e_i$. Since f_i is the heaviest edge weight of some cycle in $T + e_i$, we have that any MST of $T + e_i - f_i$ is also an MST of $T + e_i$. But $T + e_i - f_i$ is a tree itself, which is equal to T_i, and so this means T_i is an MST of $T + e_i$. This concludes the proof.

Runtime: We create the graph $G = (V, E)$ in $O(n + m)$ time and it has n vertices and m edges. The runtime of Kruskal's algorithm on this graph is $O(m \log m)$. Since $m \geq n - 1$ (otherwise, we can never make the city connected), we have that the runtime is $O(m \log m)$. Each iteration of the for-loop for computing each T_i takes another $O(m)$ time. Since we run this for-loop $O(\log m)$ times (there are $O(\log m)$ choices for e_i), this step also takes $O(m \log m)$. Thus, the overall running time is $O(m \log m)$.

Problem 4. You are given a weighted undirected graph $G = (V, E)$ with integer weights $w_e \in \{1, 2, \ldots, W\}$ on each edge e, where $W = O(1)$. Given two vertices $s, t \in V$, the goal is to find the minimum weight path (or shortest path) from s to t. Recall that Dijkstra's algorithm solves this problem in $O(n + m \log m)$ time even if we do not have the condition that $W = O(1)$. However, we now want to use this extra condition to design an even faster algorithm.

Design and analyze an algorithm to find the minimum weight (shortest) s-t path in $O(n + m)$ time.

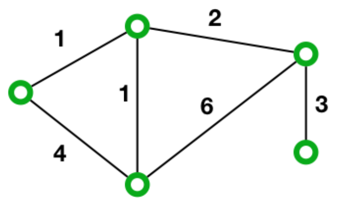
Solution. *Algorithm:* We create a new graph $G' = (V', E')$ as follows:

- Copy all vertices V in V'. Additionally for any edge $e = (u, v) \in E$, add $w_e - 1$ new vertices $z(e)_1, \ldots, z(e)_{w_e-1}$. We refer to vertices of V in V' as *original* vertices and to z-vertices as *new* vertices.
- For each edge $e = (u, v) \in E$, add the following edges to E':

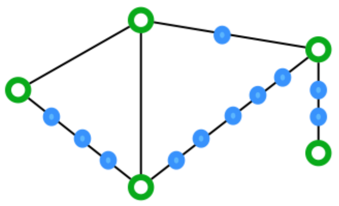
$$(u, z(e)_1), \quad (z(e)_1, z(e)_2), \quad \ldots \quad , (z(e)_{w_e-2}, z(e)_{w_e-1}), \quad (z(e)_{w_e-1}, v).$$

Note that basically in G', we are replacing each weighted edge e with weight w_e into an unweighted *path* of length w_e (i.e., with w_e edges).

See Figure 1 for an illustration.



(a) A weighted input graph with $W = 6$



(b) The unweighted graph obtained in the reduction; the original vertices are green and the new vertices are blue.

Figure 1: An illustration of the reduction

Run BFS on G' starting from the copy of vertex s and let P' be the shortest s-t path in G'. Obtain a s-t path P in G from P' by removing all new vertices and keeping the ordering of original vertices the same. Output P as the answer.

Proof of Correctness: In this reduction, any edge of weight w_e is replaced by a path of length w_e edges. This means that any path P in G with weight $w(P) = \sum_{e \in P} w_e$ is replaced with a path P' of length $w(P)$ edges.

Similarly, any path P' between two original vertices in G' that has ℓ edges will transform to a path P in the original graph with weight ℓ after we remove the new vertices from P'.

Thus, shortest (weight) s-t path in G corresponds to shortest (length) s-t path in G'. As BFS finds unweighted shortest paths correctly, the returned path P is the correct answer.

Runtime analysis: Graph G' has $n' = n + \sum_{e \in E} w_e - 1 \leq n + m \cdot W = O(n+m)$ vertices and $m' = \sum_{e \in E} w_e \leq m \cdot W = O(m)$ edges (since in both cases $W = O(1)$). Creating G' takes $O(n' + m') = O(n+m)$ time also. Running BFS on G' needs $O(n'+m')$ time which is $O(n+m)$. So the total runtime is $O(n+m)$.