CS 170 Homework 6

Due **10/11/2023, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

2 Adding Many Edges At Once

Given an undirected, weighted graph $G(V, E)$, consider the following algorithm to find the minimum spanning tree. This algorithm is similar to Prim's, except rather than growing out a spanning tree from one vertex, it tries to grow out the spanning tree from every vertex at the same time.

procedure FindMST($G(V, E)$)
  $T \leftarrow \emptyset$
  while T is not a spanning tree **do**
    Let $S_1, S_2 \dots S_k$ be the connected components of the graph with vertices V and edges T
    For each $i \in \{1, \dots, k\}$, let e_i be the minimum-weight edge with exactly one endpoint in S_i
    $T \leftarrow T \cup \{e_1, e_2, \dots e_k\}$
  return T

For example, at the start of the first iteration, every vertex is its own S_i.

For simplicity, in the following parts you may assume that no two edges in G have the same weight.

(a) Show that this algorithm finds a minimum spanning tree.

(b) Give a tight upper bound on the worst-case number of iterations of the while loop in one run of the algorithm. Justify your answer.

(c) Using your answer to the previous part, give an upper bound on the runtime of this algorithm.

Solution:

This algorithm is known as Boruvka's algorithm. Despite perhaps appearing complicated for an MST algorithm, it was discovered in 1926, predating both Prim's and Kruskal's. This algorithm is still of interest due to being easily parallelizable, as well as being the basis for a randomized MST algorithm that runs in expected time $O(|E|)$.

(a) If we add an edge e_i to T, it is because it is the cheapest edge with exactly one endpoint in S_i. So applying the cut property to the cut $(S_i, V - S_i)$ we see that e_i must be in

the (unique) minimum spanning tree. So every edge we add must be in the minimum spanning tree, i.e. this algorithm finds exactly the minimum spanning tree.

(b) The number of components in the graph with vertices V and edges T starts out at $|V|$, with one component for each vertex. If there are k components at the start of an iteration, we must add at least $k/2$ edges in that iteration: every component S_i contributes some e_i to the set of edges we're adding, and each edge we add can only have been contributed by up to two components. This means the number of components decreases by at least a factor of 2 in every iteration. Thus, the while loop runs for at most $\log |V|$ iterations.

(c) The previous part shows that at most $\log |V|$ iterations are needed. Each iteration can be performed in $O(|E|)$ time (e.g. you can compute the components in $O(|E|)$ time, and then initialize a table which stores the cheapest edge with exactly one endpoint in each component, and then using a linear scan over all edges fill out this table) giving a runtime bound of $O(|E| \log |V|)$.

3 Minimum ∞-Norm Cut

In the Minimum Infinity-Norm Cut problem, you are given a connected undirected graph $G = (V, E)$ with positive edge weights w_e, and you are asked to find a cut in the graph where the largest edge in the cut is as small as possible (note that there is no notion of source or target; any cut with at least one node on each side is valid).

Solve this problem in $O(|E|\log|V| + |V| + |E|)$ time. **Give a 3-part solution.**

Hint: Minimum Spanning Tree does not require edge weights to be positive.

Solution: Algorithm: First, negate all the edge weights in G, and pass this new graph to Kruskal's minimum spanning tree algorithm; this will give us a maximum spanning tree of the original graph. Remove the smallest-weight edge in this maximum spanning tree, and return the cut induced by its removal.

Proof of Correctness: First note that we correctly find a maximum spanning tree of the graph, since $\max_{\text{tree } T} \sum_{e \in T} w_e = \min_{\text{tree } T} \sum_{e \in T} -w_e$. It is similarly easy to see that we find a minimum infinity-norm cut in the maximum spanning tree of the graph (using any other edges from the spanning tree could not decrease the cut, since we used only the smallest possible edge).

It only remains to show that any cut of the nodes in the graph has exactly the same infinity norm in the graph overall as in the maximum spanning tree; this will prove the correctness of our algorithm (since we have already proven its correctness in the tree). To see this, we will use the cut property for maximum spanning trees (which follows immediately from the cut property for minimum spanning trees, applied to the negated graph). This property is: for any cut in the graph, its largest edge (or one of its largest edges, if the largest edge is not unique) must be contained in the maximum spanning tree. Since the infinity norm of the cut is equal to the weight of its largest edge, this means that the infinity norm of the cut in the tree is at least its infinity norm in the graph; it is also at most the infinity norm in the

graph, since no edges exist in the tree which do not exist in the graph.

Runtime Analysis: Creating a new graph with every edge negated takes $O(|V|+|E|)$ time. Once we have the maximum spanning tree, the smallest-weight edge can be found with a simple $O(|E|)$ time search; once it is removed, the nodes in the two resulting components can be enumerated with a $O(|V|+|E|)$ time traversal. So overall, we have taken linear time to convert this problem to an MST problem.

We can then run Kruskal's algorithm in $O(|E|\log|V|)$ time to find the MST. Finding the smallest-weight cut requires traversing over all edges. Hence, the final runtime is $O(|E|\log|V|+|V|+|E|)$.

Alternative solution

Sort the array of edge weights and binary search for the largest edge in the cut. For an edge with weight w_m, consider G', the graph obtained by keeping only the edges from E with weight $\leq w_m$. If G' is connected, then the answer is $\geq w_e$. Otherwise, the answer is $< w_m$.

Once you find the ∞ norm w_e of the cut, you can recover the cut by taking any connected component and its complement after removing all edges of weight $>= w_e$. Overall runtime is $O(|E|log|V|)$.

4 Firefighters

PNPLand is made of N cities that are numbered from $0, 1, \ldots N-1$, which are connected by two-way roads. You are given a matrix D such that, for each pair of cities (a, b), $D[a][b]$ is the distance of the shortest path between a and b.

We want to pick K distinct cities and build fire stations there. For each city without a fire station, the response time for that city is given by distance to the nearest fire station. We define the response time for a city with a fire station to be 0. Let R be the maximum response time among all cities. We want to create an assignment of fire stations to cities such that R is as small as possible.

Suppose the optimal assignment of fire stations to cities produces response time R_{opt}. Given positive integers N, K and the 2D matrix D as input, describe an $O(N^2 \cdot K)$ (or faster) greedy algorithm to output an assignment that achieves a response time of $R_g \leq 2 \cdot R_{\text{opt}}$. **Provide a 3-part solution.** For your proof of correctness, show that your algorithm achieves the desired approximation factor of 2.

Hint: $D[a][b]$ represents shortest (metric) distances. So you can use the triangle inequality: $0 \leq D[a][b] \leq D[a][c] + D[c][b]$ for all a, b, c.

Solution:

Algorithm Description: Start by building the fire station an arbitrary city, eg city 1. For the remaining $K-1$ fire stations, repeatedly find the city with the largest response time and build a fire station there.

Proof of Correctness: Let R_i be the largest response time creating the ith fire station

in the algorithm (for $i > 1$). Observe that at the ith step of the algorithm, the travel time between any two fire stations is $\geq R_i$. We want to show $R_K = R_g \leq 2R_{\text{opt}}$.

Suppose this algorithm results in a city v whose response time is $R_K > 2R_{\text{opt}}$. Then the travel time between every pair of fire stations is $> 2R_{\text{opt}}$.

Now consider the set S of these $K+1$ cities: the K fire centers and v. In the optimal solution, there has to a fire center w such that w is the closest fire center to two cities $a, b \in S$. Hence $D[a][b] \leq D[a][s] + D[s][b] \leq 2R_{\text{opt}}$. However, since a, b are in S, $D[a][b] > 2R_{\text{opt}}$. This is a contradiction.

Runtime Analysis: There are K iterations, and each iteration takes $O(N^2)$ time because we look at the distances between every pair of fire stations. Thus, the overall runtime is $O(N^2K)$.