601.433/633 Introduction to Algorithms **Fall 2024**
Homework #7 **Due:** November 12, 2024, 9am

Bella Lu
Partners: Jecia Mao, Luna Liu

1 Boruvka's Algorithm (40 points)

(a) (10 points) This was stated in class, but let's prove it formally. Let $G = (V, E)$ be an undirected graph and let $w : E \to \mathbb{R}_{\geq 0}$ be edge weights. Prove that if all edge weights are distinct ($w(e) \neq w(e')$ for all $e \neq e' \in E$) then there is a *unique* minimum spanning tree (MST).

For sake of contradiction, let there be two different MSTs. We can call them MST_1 and MST_2.

If they are different, this means that there is at least one edge, which we can call e_1, between two nodes, which we can call x and y, that are in MST_1 but not in MST_2. Since this edge is not in MST_2, there must be some other path that gets from x to y, which we can call p_2.

With these definitions, we can say the following:

In MST_1, weight of e_1 < total weight of p_2 because we had decided to add e_1 in MST_1. So by definition, e_1 should be the shortest path from x to y.

In MST_2, weight of e_1 > total weight of p_2 because if e_1 was smaller, then we could have swapped out an edge in p_2 with e_1 to make the path from x to y shorter. We can swap one out because adding e_1 would create a cycle.

$\therefore$ The weight of e_1 can't be both smaller and larger than the total weight of p_1, thus the two MSTs must be the same, which means that it is unique. ■

Let $G = (V, E)$ be a connected, undirected graph and let $w : E \to \mathbb{R}_{\geq 0}$ be *distinct* edge weights (w is injective). We're going to analyze yet another MST algorithm: Boruvka's MST algorithm (from 1926), which is a bit like a distributed version of Kruskal.

We begin by having each vertex mark the lightest edge incident to it. (For instance, if the graph were a 4-cycle with edges of lengths 1, 3, 2, and 4 around the cycle, then two vertices would mark the "1" edge and the other two vertices will mark the "2" edge). This creates a forest F of marked edges. (Convince yourself that there won't be any cycles!). In the next step, each tree in F marks the shortest edge incident to it (the shortest edge having one endpoint in the tree and one endpoint not in the tree), creating a new forest F'. This process repeats until we have only one tree.

(b) (10 points) Show correctness of this algorithm by proving that the set of edges in the current forest is always contained in the MST.

We can use strong induction.

Base case: We can start with a forest with all the vertices and an empty set of edges.

For the sake of induction, we can assume that for all iterations $< k$, all edges that have been added to the forest are in the MST.

For iteration k, by the algorithm, we are marking each cluster with the minimum edge, which is the min cut edge between that cluster and the rest of the graph. By the min cut theorem, this edge will be in the MST. The edges in each cluster are in the MST, as per our induction hypothesis, so adding the new edge that is the min cut between every cluster and the rest of the graph still maintains the property that every edge in the forest is in the MST.

$\therefore$ The set of edges in the current forest is always contained in the MST. ■

(c) (10 points) Show how you can run each iteration of the algorithm in $O(m)$ time with just a few runs of DFS and no fancy data structures (heaps, union-find – remember, this algorithm was from 1926!). In other words, given a current forest F of marked edges, show how to find the set of edges which consists of the shortest edge incident to each tree in F in time $O(m)$.

Step 1:

Run DFS individually on each cluster. This will return lists of edges in each cluster. We know that DFS takes O(number of edges). Adding up all the edges in all the clusters gives us m. So running DFS on each cluster takes O(edges in cluster1 + edges in cluster2 + ...) = O(m).

Step 2:

We can compare the lists of edges in each cluster with the list of all the edges in the graph, and pick out which edges aren't in any cluster. This will return a list of all the between-cluster edges. This takes O(m) because we are simply traversing and comparing the lists, which has O(m) elements total.

Step 3:

We can go through all the between-cluster edges using DFS, and figure out the minimum edge between each cluster. For example, create a mapping of which cluster each vertex of each between-cluster edge belongs to, then update the minimum edge between each cluster if we find one that is smaller. Creating mappings would take O(m) and traversing the list would also take O(m).

Thus, the algorithm takes O(m) with just a few runs of DFS each iteration.

(d) (10 points) Prove an upper bound of $O(m \log n)$ on the running time of this algorithm.

We have proved in part (c) that each iteration takes O(m).

There are $\log n$ number of iterations because the number of clusters halves by each time, since in each iteration we are finding the minimum edge between two clusters, and merging the two clusters.

So every time we add an edge, at least two clusters become one.

Thus, the total runtime has an upper bound of $O(m \log n)$.

2 Clustering (30 points)

Consider the following problem. You are given a graph $G = (V, E)$ and a length function $\ell : E \to \mathbb{R}_{\geq 0}$. You should output a clustering (i.e., a partition) $(C_1, C_2, \ldots, C_k)$ of V into k clusters which maximizes the minimum distance between clusters. More formally, you should maximize $\min_{i,j \in [k], i \neq j} d(C_i, C_j)$, where $d(C_i, C_j) = \min_{x \in C_i, y \in C_j} d(x, y)$ and where $d(x, y)$ is the shortest-path distance (under length function ℓ). Design an algorithm for this problem which runs in $O(m \log n)$ time, and prove correctness and running time.
Hint: Think about MSTs and Kruskal's algorithm

Algorithm
We can run Kruskal's algorithm by adding minimum edges until we have added n-k edges, in which we will have k clusters.

Correctness
Let's call our algorithm's solution S_1. In S_1, let's assume that there are two vertices that are inside a cluster, that has $path_{x-y}$. By definition of our algorithm, every between-cluster edge in S_1 is larger than every edge in $path_{x-y}$ because of the ordering that we added the edges.

For the sake of contradiction, let's assume that there exists another more optimal solution, which we can call S_2, where x and y are in different clusters. Since these two solutions are different, there must exist such two vertices. Since x and y are in different clusters, there must exist some edge, lets call e that leaves the cluster, and by our algorithm, every edge in $path_{x-y}$ is smaller than the between-cluster edges in S_1. Thus, the edge that leaves the cluster in S_2, e, is smaller than the minimum between-cluster edge in S_1. This proves that our algorithm maximizes the minimum between-cluster edge.

$\therefore$ When we terminate at k clusters, the algorithm partitions the graph such that it maximizes the minimum distance between clusters.

Runtime

As proved in class, Kruskal's algorithm takes $O(m \log n)$ (includes sorting). In our algorithm, we are simply doing Kruskal's algorithm with an early termination, so the overall runtime is $O(m \log n)$.

3 Matroids (30 points)

(a) (10 points) Let U be a finite set and let $k \geq 0$ be an integer. Let $\mathcal{I} = \{S \subseteq U : |S| \leq k\}$. Prove that $(U, \mathcal{I})$ is a matroid.

Property 1: If $k \geq 0$, then $|S| = 0$ is included. The only set that has cardinality equal to 0 is the empty set.

Property 2: Let S' be a subset of $S \in \mathcal{I}$. $|S'| \leq |S|$ because we can create an injective mapping from every element in S' to itself. Since S' s included in S, this means that the cardinality of S must be at least as big as S'. Since $S \in \mathcal{I}$, $|S| \leq k$ and since $|S'| \leq |S|$, this means that $|S'| \leq k$, which means that $S' \in \mathcal{I}$.

Property 3: Let $|S_2| > |S_1|$. This means that there cannot be a bijection between the two. There must be at least one element that is in $|S_2|$ that is not in $|S_1|$ because it has more elements.

Since all 3 properties hold, $(U, \mathcal{I})$ is a matroid.

(b) (20 points) Let $(U, \mathcal{I})$ be a matroid, and let $w : U \to \mathbb{R}^+$ be an assignment of weights to elements. We know that the greedy algorithm will return a maximum-weight independent set. But suppose we are also given some $I \in \mathcal{I}$, and want to return the maximum-weight independent set *that contains I*. Modify the greedy algorithm to solve this problem, and prove that your algorithm is correct. You do not need to analyze running time, but your algorithm should need only a polynomial amount of time and calls to the independence oracle.

Algorithm

$F = I$

$U' = U \setminus I$

Sort U' by weight (largest to smallest)

For each $u \in U'$ in sorted order:

    If $F \cup \{u\} \in \mathcal{I}$, add u to F

return F

Correctness

I is an independent set because in the question statement, we are given that $I \in \mathcal{I}$.

Since we are only adding u to F if $F \cup \{u\} \in \mathcal{I}$ in our algorithm, every edge we add maintain the property that our current set is still an independent set.

To prove that the algorithm will return the maximum-weight independent set that contains I, let's define the following list:

Algorithm output: $F = \{F_I | F_{\setminus I}\}$, which is partitioning the list F into 2 parts, a list containing edges in I and a list edges excluding I

Some other output: $F' = \{F'_I | F'_{\setminus I}\}$, partitioned in the same way, so that $F_I = F'_I$

For $F_{\setminus I} = \{f_1, f_2, ..., f_r\}$, $w(f_i) \geq w(f_{i+1})$ for all i (order added by greedy)

For $F'_{\setminus I} = \{e_1, e_2, ..., e_r\}$, $w(e_i) \geq w(e_{i+1})$ for all i

We want to show that $w(f_i) \geq w(e_i)$ for all i

For the sake of contradiction, let j be the smallest integer such that $w(f_j) < w(e_j)$

Let $F_1 = \{f_1, ..., f_{j-1}\}$ and $F_2 = \{e_1, ..., e_j\}$

We know that $|F_2| > |F_1|$, so by the augmentation property, there is some element $e_z \in F_2 \setminus F_1$ such that $F_1 \cup \{e_z\} \in \mathcal{I}$

This means that $w(e_z) \geq w(e_j) > w(f_j)$.

Since $w(e_z) > w(f_j)$, greedy would add e_z next, and not f_j, which is a contradiction.

Thus, since $w(f_i) \geq w(e_i)$ for all i, and since $F_I = F'_I$, the output return by our algorithm, F, is the maximum-weight independent set containing I.