CS170Hw6

Trustin Nguyen

October 12, 2024

Study Group

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. The distances $D[a][b]$ are metric, so you can assume that the triangle inequality always holds: $0 \leqslant D[a][b] \leqslant D[a][c] + D[c][b]$ for all a, b, c.

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.

In this problem, we'll devise a greedy algorithm to solve this problem. However a greedy approach will not always give the optimal solution. Instead, we'll just aim to get a solution that is "good enough".

We can formalize the problem as follows: 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 $\mathcal{O}(N^2 \cdot K)$ (or faster) greedy algorithm to output an assignment that achieves a response time of $R_g \leqslant 2 \cdot R_{opt}$.

Provide a 3-part solution. For your proof of correctness, show that your algorithm achieves the desired approximation factor of 2.

Answer. Algorithm: We pick a fire station iteratively. Now for every vertex, we calculate the minimum length of all edges attached to it. Once we have these minimum lengths, we pick the vertex that contains the maximum of these minimum lengths to be a fire station. Now we repeat the process until we have K fire stations.

Runtime Analysis: Each time we choose a fire station, we iterate over all the edges. We do this K times, so the runtime is $\mathcal{O}(K \cdot |E|) = \mathcal{O}(K \cdot N^2)$.

Proof of correctness: Let us denote the optimal arrangement of fire stations as F_{Opt}. Then the longest response time of F_{Opt} is R_{Opt} denote the edge with length R_{Opt} in F_{Opt} as (f_1, f_2).

We know that one of f_1, f_2 must be a fire station, wlog say this is f_1. This is because since there must be at least one fire station, f_1 has a direct path to that fire station and by triangle inequality, traveling through two paths is worse than traveling through one.

Now suppose that g_1 is a fire station in G, our greedy solution, that is not a fire station in F_{Opt}. We will show that the path from g_1 to any city g_2 is no worse than $2 \cdot R_{Opt}$.

- If g_2 is a fire station in G, then we are done, as the response time is $0 \leqslant 2 \cdot R_{Opt}$. So assume otherwise.
- If g_2 is a not a fire station in F_{Opt}, then we know that the response time for g_2 is at worst R_{Opt}. We also know that since g_1 is not a fire station in F_{Opt}, the response time for g_1 is at worst R_{Opt} also. Then traveling from $g_1 \to f_1 \to g_2$ is of path length at most $2 \cdot R_{Opt}$.
- Now for the case when g_2 is a fire station of F_{Opt}. Since g_1 is not a fire station in F_{Opt} it must be within distant R_{Opt} to some fire station f' in F_{Opt} and therefore within R_{Opt} distance of a city in G. Since g_1 was selected to be a fire station before g_2 in G, then the minimum edge out of g_1 which is $\leqslant R_{Opt}$ is greater than the minimum edge out of g_2. So g_2 must be within R_{Opt} of some fire station in G or if its within R_{Opt} response time to a city, we know we can connect it to the fire station f_1 through another R_{Opt} response time. So g_2 has $\leqslant 2R_{Opt}$ response time

From the three cases, we conclude that the worst case response time is $2R_{\mathtt{Opt}}$ given our algorithm.

Updating a MST

You are given a graph $G = (V, E)$ with positive edge weights, and a minimum spanning tree $T = (V, E')$ with respect to these weights; you may assume G and T are given as adjacency lists. Now suppose the weight of a particular edge $e \in E$ is modified from $w(e)$ to a new value $\hat{w}(e)$. You wish to quickly update the minimum spanning tree T to reflect this change, without recomputing the entire tree from scratch.

There are four cases. In each, give a description of an algorithm for updating T a proof of correctness, and a runtime analysis for the algorithm. Note that for some of the cases these may be quite brief. For simplicity, you may assume that no two edges have the same weight (this applies to both w and $\hat{w}$).

(a) $e \in E'$ and $\hat{w}(e) < w(e)$.

Answer. Algorithm: We would do nothing here.

Runtime Analysis: The runtime is them $O(1)$.

Proof of correctness: We see that the new tree with where e is replaced with a smaller weight has a smaller total edge weight then our previous MST. By the transitive property, changing the edge weight of e as described will still mean E' the edge set of the MST.

(b) $e \notin E'$ and $\hat{w}(e) < w(e)$

Answer. Algorithm: We would run DFS on the graph, where in the pre-order, we keep track of the current path taken by pushing the vertex to the path. In the post-order, we would pop the last vertex. When DFS finds a node that we have already visited, we will know that it has found a cycle. Then our recorded path has that cycle. Loop through the cycle and find the edge of largest weight to remove.

Runtime Analysis: The runtime of DFS is $O(|V| + |E|) \to O(|V|)$ since $|E| = |V| - 1$.

Proof of correctness: Let e be the edge of largest weight that we had removed, e_1, e_2 be the endpoints. Then for every e' in the cycle that we did not remove, let e'_1, e'_2 be their vertices. Now for every e' in the cycle, remove e' and e from the graph. Wlog say that e_1, e'_1 are in the same connected component and e_2, e'_2 are in the same connected component. We see that the edge with smaller weight must be in the MST. Well, over all these cuts, edge e will never have the smaller weight. So it must be that all e' are in the MST and not e.

(c) $e \in E'$ and $\hat{w}(e) > w(e)$

Answer. Algorithm: We would first remove e from the MST and try to find a new edge with the cut property. This means finding the 2 connected components we are left with, probably through DFS. Then for each edge that bridge the connected components, we find the edge with minimum weight to add to the MST.

Runtime analysis: The runtime would be $O(|V|^2)$ because DFS takes $O(|V| + |E|) = O(|V|)$, but in worst case, each connected component has $|V|/2$ vertices and in a connected graph, there would be $(|V|/2)^2$ edges crossing the connected component. So we would have to check $O(|V|^2)$ edges.

Proof of correctness: By the cut property, we know that the edge of minimum weight across a cut must be in the MST. In our algorithm, we made a cut, so the new edge must be the one chosen by our algorithm. Note that all other edges can stay the same because any cut we make with a different edge will have the endpoints of e in the same connected component. So e will have no impact.

(d) $e \notin E'$ and $\hat{w}(e) > w(e)$

Answer. Algorithm: We would do nothing here.

Runtime Analysis: The runtime is $O(1)$.

Proof of correctness: Doing nothing still results in an MST. Since e was not the minimum edge in any cut, then increasing its weight will still mean that it is not the minimum edge in any cut. So we will always choose some other edge over e to be in the MST.

Minimum Spanning k-Forest

Given a graph $G(V, E)$ with non-negative weights, a spanning k-forest is a cycle-free collection of edges $F \subseteq E$ such that the graph with the same vertices as G but only the edges in F has k connected components. For example, consider the graph $G(V, E)$ with vertices $V = \{A, B, C, D, E\}$ and all possible edges. One spanning 2-forest of this graph is $F = \{(A, C), (B, D), (D, E)\}$, because the graph with vertices V and edges F has components $\{A, C\}, \{B, D, E\}$.

The minimum spanning k-forest is defined as the spanning k-forest with the minimum total edge weight. (Note that when $k = 1$, this is equivalent to the minimum spanning tree). In this problem, you will design an algorithm to find the minimum spanning k-forest. For simplicity, you may assume that all edges in G have distinct weights.

(a) Define a j-partition of a graph G to be a partition of the vertices V into j (non-empty) sets. That is, a j-partition is a list of j sets of vertices $\Pi = \{S_1, S_2, \dots, S_j\}$ such that every S_i includes at least one vertex, and every vertex in G appears in exactly one S_i. For example, if the vertices of the graph are $\{A, B, C, D, E\}$, one 3-partition is to split the vertices into the sets $\Pi = \{\{A, B\}, \{C\}, \{D, E\}\}$.

Show that for any j-partition Π of a graph G, if $j > k$ then the lightest edge crossing Π must be in the minimum spanning k-forest of G.

Answer. Suppose for contradiction that e was of minimum weight that crosses the j-partition. Then in the MST, we have some set of edges E' that were added from the j partition such that we get a k-forest. Note that e is not in E', the set of edges that we added.

Now add e into the k-forest as an edge. If there is a cycle, we know that the number of connected components does not decrease.

There is some e' in E' such that e' has endpoints in the same connected component that e does. Then since e has smaller weight than e', it is guaranteed to be part of the MST, and we would remove some maximum weight edge of that cycle. We are then left with a minimum spanning k-forest

Now if adding in e does not create a cycle, then the number of connected components is then $k - 1$. But since e has least weight out of all $e' \in E'$. We know that we can remove some other edge $e' \in E'$ that crosses the j-partition. So now we are back to k connected components, each containing a tree. The total edge weight is smaller because e was of minimum weight, so this shows that e must be part of the minimum spanning k-forest.

(b) Give an efficient algorithm for finding the minimum spanning k-forest. Please give a 3-part solution.

Answer. Algorithm: First, sort all edges by weight, keep this in an array E_{sorted}. Keep track of the connected components using a union-find data structure. Start by popping elements from E_{sorted}. Check if the endpoints are part of the same root in the union-find data structure. If not, add the edge to the MST by union in the union-find. Repeat this process such that we have run the union $|V| - k$ times.

Runtime Analysis: It takes $|E| \log |E|$ time to sort the edges. Then we iterate through the $|E|_{\text{sorted}}$ list which takes $|E|$ operations and each time, we do a check to see if the vertices share a root $O(\alpha(n))$ time. So total runtime is $O(|E| \log |E|)$.

Proof of correctness: By the previous part, we know that the minimum edge across a cut must be in the MST. So in our algorithm, we take the minimum edge across a cut and keep adding edges until we get k connected components. So the algorithm must yield a minimum spanning k-forest.

Copper Pipes

Bubbles has a copper pipe of length n inches and an array of non-negative integers that contains prices of all pieces of size at most n. He wants to find the maximum value he can make by cutting up the pipe and selling the pieces. For example, if length of the pipe is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6).

price	1	2	3	4	5	6	7	8
length	1	5	8	9	10	17	17	20

Give a dynamic programming algorithm so Bubbles can find the maximum obtainable value given any pipe length and set of prices. Clearly describe your algorithm and analyze its runtime (proof of correctness not required).

Answer.

```
def maxGain(prices):
    # the price for a pipe of length 0 is 0
    m_prices = [0] + prices

    dp ← [0] + prices
    for i in range(1, len(prices) + 1):
        for j in range(i):
            dp[i] = max(dp[i], dp[j] + dp[i - j])

    return dp
```

The algorithm checks for a pipe of length i, the best price over all possible cuts that include a subpipe of length j. The price of a segment of length j and the best selling price for a pipe of length $i - j$ is added to see the best price for a pipe cut with one length being j. So this gives us all combinations of cuts. We take the max over those combinations to get the max selling price.

The runtime is $O(n^2)$ where n is the length of prices.