CS 170 Efficient Algorithms and Intractable Problems
February 23, 2016 Liam Leahy

HW 5

Due February 22, 11:59pm

Partner: Stephen Qu

1. (20 pts.) Short Questions

1. **(10 pts.)** Let G be a connected undirected graph with positive lengths on all the edges. Let s be a fixed vertex. Let $d(s,v)$ denote the distance from vertex s to vertex v, i.e., the length of the shortest path from s to v. If we choose the vertex v that makes $d(s,v)$ as small as possible, subject to the requirement that $v \neq s$, then does every edge on the path from s to v have to be part of every minimum spanning tree of G?
2. **(10 pts.)** The same question as above, except now no two edges can have the same length.

Solution:

(a) No, consider the simplest case. For example, there could be a graph of three nodes that is fully connected and every edge has weight 1. In this case, the MST would be decided via a tie-breaking scheme and thus there is no guarantee any edge is in the MST.

(b) Yes. First of all, v is necessarily a neighbor of s and it is the closest neighbor. This is because if v is close but not a neighbor, then there is necessarily a vertex closer to s than v, since there are no negative edges. It is the closest neighbor because if it is not, reassign it to be the closest neighbor. Call e the edge from s to v. e is the best edge to use to connect s to the MST so long as it does not create a cycle. Now the only case in which it would not be added to the MST is if it creates a cycle. This is impossible as no other edge from s has been considered yet, so it cannot be the case that it creates a cycle as the cycle would have to go through s.

2. (20 pts.) Preventing Conflict

A group of n guests shows up to a house for a party, and any two guests are either friends or enemies. There are two rooms in the house, and the host wants to distribute guests among the rooms, breaking up as many pairs of enemies as possible. The guests are all waiting outside the house and are impatient to get in, so the host needs to assign them to the two rooms quickly, even if this means that it's not the best possible solution. Come up with an efficient algorithm that breaks up at least half the number of pairs of enemies as the best possible solution, and prove your answer.

Solution:

Main Idea: Put any person in the room with the fewest number of enemies for said person.

Pseudocode:

Algorithm 1 Put people into rooms to guarantee at least half of the enemies are partitioned.

```
function PARTITION_PARTY_PEOPLE($P$)
    choose a random person $p$ and put him in a room
    $P -= p$
    for person $p \in P$ do
        enemiesInA, enemiesInB = 0,0
        for enemy in p.enemies do
            if enemy in room A then
                enemiesInA += 1
            else
                enemiesInB += 1
        if enemiesInA > enemiesInB then
            put $p$ in roomA
        else
            put $p$ in roomB
        $P -= p$
    return partitioned roomA and roomB
```

Proof: Proof by induction on the number of people n.

If there are no enemies, trivial.

If there is at least one enemy pair, then we have put them in other rooms.

Assume it successfully partitions for the first k people such that each time at least half of the people that are put into rooms are partitioned in such a way that at least half of the enemy connections are broken by the rooms.

Consider adding one more person $k+1$. It is either the case that there are more enemies in one room or there are not. If there are, we break some number (greater than one) of enemy connections by putting this person in the room with the smaller number. We say there is a set E of enemy connections. If there are the same number of enemies in each room for the $k+1$th person then we are adding some number of edges and "diconnecting" half of them, this maintains the property that at least half are connected. If there is more on one side then the same property holds if we choose to insert into the side with fewer enemies as we "disconnect" the side with more enemies. □

Running Time: $O(N^2)$.

Justification: For every iteration through N we consider at most N connections. Another way to think about it is we construct a graph of the enemy pairs. There are at most N^2 edges and N vetices. We run through the graph linearly in $O(|V|+|E|)$ time which is in $O(N^2)$.

3. (20 pts.) Graph Subsets

Let $G = (V, E)$ be a connected, undirected graph, with edge weights $w(e)$ on each edge e. Some edge weights *might be negative*. We want to find a subset of edges $E' \subseteq E$ such that $G' = (V, E')$ is connected, and the sum of the edge weights in E' is as small as possible.

(a) Is it guaranteed that the optimal solution E' to this problem will always form a tree?

(b) Does Kruskal's algorithm solve this problem? If yes, explain why in a sentence or two; if no, give a small counterexample.

(c) Describe an efficient algorithm for this problem. Be concise. You should be able to describe your algorithm in one or two sentences. (You donâ€™t need to prove your algorithm correct, justify it, show pseudocode, or analyze its running time.)

Solution:

(a) No, consider a graph that is entirely composed of negative edges and it is fully connected. Using this would result in taking every edge in the graph, which (since it is fully connected) would result in this graph.

(b) No, Kruskal's doesn't allow cycles but the optimal solution may have cycles. For example, consider any graph with all negative edges, Kruskal's will find the optimal solution without cycles but the optimal solution includes all the edges.

(c) Dump all the negative edges into the solution and then perform Kruskal's.

4. (20 pts.) Timesheets Suppose we have N jobs labeled $1, \ldots, N$. For each job, there is a bonus $V_i \geq 0$ for completing the job, and a penalty $P_i \geq 0$ per day that accumulates for each day until the job is completed. It will take $R_i \geq 0$ days to successfully complete job i.

Each day, we choose one unfinished job to work on. A job i has been finished if we have spent R_i days working on it. This doesn't necessarily mean you have to spend R_i consecutive days working on job i. We start on day 1, and we want to complete all our jobs and finish with maximum reward. If we finish job i at the end of day t, we will get reward $V_i - t \cdot P_i$. Note, this value can be negative if you choose to delay a job for too long.

Given this information, what is the optimal job scheduling policy to complete all of the jobs?

Solution:

Main Idea: We do not need to worry about V_i as this is basically a "fixed sum" that only decreases each time step. This means that we simply want to minimize the loss that we obtain from the total time required. Additionall,y, the most efficient alfgorithm should choose to start and finish a job before starting another. This is because we only lose score for each time step and we are minimizing the loss. Sort the elements using the comparator between any two elements i, j: $P_j \cdot R_i$ vs. $P_i \cdot R_j$ choosing i or j based on whichever term with subscript for R is smaller.

Pseudocode:

Algorithm 2 Find the best ordering for the time jobs.

```
1: function BEST_JOB_ORDERING($J[1, \ldots, N]$)
2:     $J \leftarrow$ SORT_WITH_COMPARATOR($P_j \cdot R_i$ vs. $P_i \cdot R_j$)
3:     return $J$
```

Proof: Consider any two jobs J_i and J_k. There are only two possibilities for their respective ordering. We want to minimize the total cost by choosing the best ordering. The cost of J_i going first is given by $P_i \cdot R_i + P_j \cdot (R_i + R_j)$ where P is the penalty and R is the time for each i and j. The reverse argument follows for the reverse ordering, yielding: $P_j \cdot R_j + P_i \cdot (R_j + R_i)$. We want the minimum of these expressions as this is the superior ordering. Notice after expanding these expressions that the difference between them is merely $P_j \cdot R_i$ for the first and $P_i \cdot R_j$ for the second. This implies that we simply want to minimize this value. You can think of this value as simply the additional cost of either job waiting for the other job to complete before it can start.

This implies that any ordering is better if the minimum of these values is chosen to come first. Therefore, we use binary sort comparing any two elements using this comparison and choosing the job J_i to come first if $P_j \cdot R_i$ is less than its repsective value. This will guarantee an ordering that is optimal with cost minimization, which is all that matters. $\square$

Running Time: $O(N \log N)$.

Justification: We can run something like merge sort on the list of jobs using a simple comparator for the ordering. There is nothing special about the comparator as it takes a constant time to evaluate with respect to the problem size, N.

5. (20 pts.) A greedy algorithm – so to speak The founder of LinkedIn, the professional networking site, decides to crawl LinkedIn's relationship graph to find all of the *super-schmoozers*. (He figures he can make more money from advertisers by charging a premium for ads displayed to super-schmoozers.) A *super-schmoozer* is a person on LinkedIn who has a link to at least 20 other super-schmoozers on LinkedIn.

We can formalize this as a graph problem. Let the undirected graph $G = (V, E)$ denote LinkedIn's relationship graph, where each vertex represents a person who has an account on LinkedIn. There is an edge $\{u, v\} \in E$ if u and v have listed a professional relationship with each other on LinkedIn (we will assume that relationships are symmetric). We are looking for a subset $S \subseteq V$ of vertices so that every vertex $s \in S$ has edges to at least 20 other vertices in S. And we want to make the set S as large as possible, subject to these constraints.

Design an efficient algorithm to find the set of super-schmoozers (the largest set S that is consistent with these constraints), given the graph G.

Hint: There are some vertices you can rule out immediately as not super-schmoozers.

Solution:

Main Idea: Continually remove all the vertices (and their corresponding edges) that have fewer than 20 edges until all vertices in the graph have at least 20 edges.

Pseudocode:

Algorithm 3 Find the best set of all super-schmoozers.

```
1:  function FIND_S($G$)
2:      contains = TRUE
3:      while contains do
4:          contains = FALSE
5:          $R = []$
6:          for all $v \in V$ do
7:              if $v$ has fewer than 20 edges then
8:                  append $v$ to $R$
9:                  contains = TRUE
10:         remove all marked $v$ in $R$ from $V$
11:     return what remains of $G$
```

Proof: Every iteration, we remove all the vertices that cannot possibly be in S. If any vertex has fewer than 20 edges, we know that it cannot possibly be in S.

The first time that we have a graph that is only composed of vertices that have more than 20 edges, we know we have a solution for S.

This is optimal because it is necessarily the case that removing the vertices with fewer than 20 edges is necessary as they cannot possibly be in S. The same can be said with the next iteration, and recursively we find the first assignment. This is the largest assignment as it is the first to occur. This is because every iteration, we only eliminate vertices that are *necessarily not in S*. This means that the first time we have a solution, up until that point, we have only removed vertices not possibly in S. □

Running Time: $O(|V|^2)$.

Justification: This algorithm simply iterates through every vertex every iteration of the while loop. In the worst case, we remove some small number (say, 1) vertices from the graph. The worst-case running time for this, therefore, is $|V|$ for each $|V|$ while-loop iteration. Which results in $O(|V|^2)$ running time.