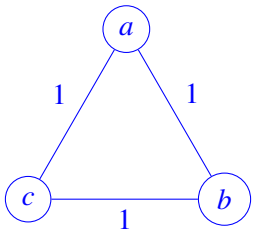
CS 170 Efficient Algorithms and Intractable Problems
Spring 2016 Alessandro Chiesa and Umesh Vazirani **HW Solution 5**

Due February 22, 11:59pm

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?

   Solutions: False. Consider the following counterexample:

   Take $s = a$. Both $v = b$ and $v = c$ minimize $d(a,v)$, but neither edge is part of every MST: for instance, (a,b) and (b,c) form a minimum spanning tree that does not contain (a,c).

2. **(10 pts.)** The same question as above, except now no two edges can have the same length.

   Solutions: True. First let's analyze the the definition. We need to find a vertex v that minimizes $d(s,v)$. Because the edge weights are positive, v has to be a neighbor a s. Or in other words, part (h) claims the lightest edge that is incident on s has to part of every minimum MST. Let us call this edge e. Assume, for contradiction, that e is not part of some MST T. Let us add e to T. This creates a cycle, which goes through e to s and exit s through another edge e'. We now remove e'. Removing an edge from a cycle keeps the graph connected and a tree. This creates a tree which is lighter than T which contradicts our assumptions that T is a MST.

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.

Solutions:

Main Idea: Let guests be nodes in a graph $G = (V,E)$ and there is an edge between each pair of enemies. The number of conflicts prevented after partitioning nodes into two non-intersecting sets A and B (also called a cut) is the number of edges between A and B. We assign nodes to the sets one by one in any order. A node

v not yet assigned would have edges to nodes already in sets A or B. We greedily assign it to the set where it has a smaller total number of edges to. This cuts at least half of the total edges.

Pseudocode:

1. Initialize empty sets A and B
2. For each node $v \in V$:
3. Initialize $n_A := 0$, $n_B := 0$
4. For each $\{v,w\} \in E$:
5. Increment n_A or n_B if $w \in A$ or $w \in B$, respectively
6. Add v to set A or B with lower n_A or n_B, break ties arbitrarily
7. Output sets A and B

Proof of Correctness: In each iteration, when we consider a vertex v, its edges are connected to other vertices already in these three disjoint sets: A, B, or the set of not-yet-assigned vertices. Let the number of edges in each case be n_A, n_B and n_X respectively. Suppose we add v to A, then n_B edges will be cut, but n_A edges can never be cut. Since $\max(n_A, n_B) \geq \frac{n_A+n_b}{2}$, each iteration will cut at least $\frac{n_A+n_b}{2}$ edges, and in total at least $\frac{|E|}{2}$ of the edges will be cut. Let $\texttt{greedycut}(G)$ be the number of edges cut by our algorithm, and $\texttt{maxcut}(G)$ be the best possible number of edges cut. Thus

$$\frac{\texttt{greedycut}(G)}{\texttt{maxcut}(G)} \geq \frac{\texttt{greedycut}(G)}{|E|} \geq \frac{|E|/2}{|E|} \geq \frac{1}{2}$$

Running Time: Each vertex is iterated through once, and each edge is iterated over at most twice, thus the runtime is $O(|V|+|E|)$.

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.

1. Is it guaranteed that the optimal solution E to this problem will always form a tree?
   Solutions: No. Consider a graph with every edge present (a complete graph), each with weight -1. Then the solution is clearly taking every edge of G, and this is definitely not a tree.
2. Does Kruskal's algorithm solve this problem? If yes, explain why in a sentence or two; if no, give a small counterexample.
   Solutions: No. As in the example above, the answer is not always a tree. Kruskals algorithm always returns a tree.
3. Describe an efficient algorithm for this problem. Be concise. You should be able to describe your algorithm in one or two sentences. (You dont need to prove your algorithm correct, justify it, show pseudocode, or analyze its running time.)
   Solutions: Add all the negative weight edges to E and contract the edges as you add them. Run Kruskals Algorithm to determine what remaining edges to add to make sure E is connected.

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?

Solutions:

Main Idea: Sort the jobs in order of decreasing P_i/R_i and allocate the first R_i available days to that job. Repeat until all N jobs have been completed.

Proof of correctness: Because interleaving the work for several jobs will delay all the jobs' completion dates, interleaving will necessarily have a higher penalty than a contiguous scheduling, and cannot be a part of any optimal solution. Therefore, we focus our attention only on orders where each job is scheduled on contiguous days: once you start a job, you complete it before starting any other job. Also, since we must complete all jobs, the V_i values are irrelevant: the base reward V_i is always received, no matter what order we schedule the jobs. Therefore, we can treat all the V_i as zero and just minimize the total penalties accrued, and this won't change the optimal solution.

To prove that our order is optimal, we will use the following swapping rule:

- Suppose the jobs are numbered $1, 2, \ldots, n$ in the order they appear in the ordering, and suppose $P_j/R_j \leq P_{j+1}/R_{j+1}$. Then we can swap jobs j and $j+1$.

We'll first prove that this modification, if applied to any order, will never make things worse (it will never increase the penalty). Why? Well, the penalty for the days when jobs j and $j+1$ are being done is

$$\rho = R_j P_j + (R_j + R_{j+1})P_{j+1} + (R_j + R_{j+1})(P_{j+2} + P_{j+2} + \cdots + P_n),$$

before the swap. After the swap, the penalty for that time period becomes

$$\rho' = (R_j + R_{j+1})P_j + R_{j+1}P_{j+1} + (R_j + R_{j+1})(P_{j+2} + P_{j+2} + \cdots + P_n).$$

Notice that

$$\rho' = \rho + R_{j+1}P_j - R_j P_{j+1}.$$

Now if $P_j/R_j \leq P_{j+1}/R_{j+1}$, then $R_{j+1}P_j \leq R_j P_{j+1}$, so $\rho' \leq \rho$. The penalty for the other days is unaffected by the swap. This proves that the swapping rule above can never increase the penalty.

Now let's prove that our algorithm generates an optimal ordering. If we start from any order, then we can repeatedly apply the swapping rule above until the jobs are ordered by decreasing P_i/R_i-value, i.e., until $P_1/R_1 \geq P_2/R_2 \geq \cdots \geq P_N/R_N$. Basically, we just apply bubble sort: take the job with largest P_i/R_i and swap it forward until it is at the start of the order; then take the job with second-largest P_i/R_i and repeatedly swap to move it forward until it as second in the order; and so on. We can see that each step, the swap moves a job with larger P_i/R_i-value forward in front of a job with smaller P_i/R_i-value, so is allowed by the swapping rule. Therefore, the swapping rule above allows us to transform any order into the one output by our algorithm, without increasing the total penalty. As a consequence, the order produced by our algorithm must be optimal.

Running Time: The proof of correctness uses bubblesort (comparing two adjacent elements) to show that the sorted order is optimal. In reality we can use a more efficient sorting algorithm, to get a runtime of $O(N \log N)$.

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.

Solutions:

Main idea.

The basic idea here is that we iteratively remove non-schmoozers until we reach a fixed point. Any node whose degree is below 20 is certainly a non-schmoozer. We keep a worklist (implemented as a set) of pending non-schmoozers. In each iteration we remove a non-schmoozer from the worklist, delete it, adjust the degree of its neighbors, and add them to the worklist if their degree has fallen below 20.

Pseudocode.

1. Set $d[v] := 0$ for each $v \in V$.
2. For each edge $\{u, v\} \in E$:
3. Increment $d[u]$. Increment $d[v]$.
4. Initialize a set W as $W := \{v \in V : d[v] < 20\}$.
5. While W is non-empty:
6. Remove a vertex from W; call it v.
7. For each $\{v, w\} \in E$:
8. Decrement $d[w]$.
9. If $d[w] < 20$ and $w \notin W$, add w to W.
10. Remove v from the graph.
11. Output the set of nodes left in the graph (they are the super-schmoozers).

Proof of correctness.

Lemma: At each iteration of the while loop, for each vertex v that has not yet been deleted, $d[v]$ = the degree of v (in the graph that remains).

Proof: This invariant can be easily proven by induction on the number of iterations of the loop. Whenever we delete a vertex, we decrement each of its neighbors to reflect the change in their degree.

Claim: After the algorithm ends, we've identified a subgraph G' such that every user in G' has degree at least 20, in that subgraph. In other words, the algorithm outputs a valid set of vertices who satisfy the requirements to be super-schmoozers.

Proof: At the beginning of every iteration of the loop, every node either has degree at least 20, or is in the worklist W. Nodes in the worklist could not possibly be super-schmoozers. This is true before the first iteration because that's how line 4 initialized the worklist. After that, a node's degree can only change when one of its neighbors is removed, and whenever a node is deleted we check all of its neighbors' degrees. Therefore, after the loop, when the worklist is empty, every remaining vertex [if any] will have degree ≥ 20, and therefore can be validly labelled a super-schmoozer.

Claim: Everyone who can be validly labelled as a super-schmoozer is included in the output of the algorithm.

Proof: Let S be the largest set of vertices that can be validly labeled as super-schmoozers. Then it is an invariant that $d[s] \geq 20$ and $s \notin W$, throughout the algorithm, for all $s \in S$. This is true before the first iteration because of how line 4 initialized the worklist. Also, if it is true before one iteration of the loop, it remains true after that iteration. In particular, consider an iteration where we remove v from the worklist. By the contrapositive of the inductive hypothesis and using the fact that we had $v \in W$ at the start of this iteration, it follows that $v \notin S$. Also, for each neighbor w of v, if $w \in S$, by the inductive hypothesis w had an edge to ≥ 20 other members of S; since $v \notin S$, after deleting v it still has an edge to ≥ 20 other members of S. Therefore, the invariant remains true after this iteration of the loop, and the claim follows by induction.

The first claim shows that the algorithm's output is not "too large", and the second claim shows that the algorithm's output is not "too small". This implies that our algorithm finds the largest possible set of super-schmoozers, as desired.

Running time. $O(|V| + |E|)$. We examine each vertex twice (in lines 1 and 4), and do a constant amount of work per vertex in each case. Each vertex is inserted into W at most once, so the number of iterations of the loop in lines 5–10 is at most $|V|$, and each iteration takes $O(1)$ time, ignoring the inner loop at lines 7–9. (Notice that using a HashSet you can insert, remove and lookup items in $O(1)$ time.) Also, the inner loop at lines 7–9 examines each edge at most once, when we remove one of its endpoints, and we we do a constant amount of work per edge in that case. This accounts for all of the work done in this algorithm, and we can see that we do $O(1)$ work per vertex plus $O(1)$ work per edge. Hence, the running time is $O(|V| + |E|)$.