CS 170 HW 4

Due on 2017-02-19, at 11:59 pm

1 (★) Study Group

List the names and SIDs of the members in your study group.

2 (★) Short Answer Questions

(a) Let G be a dag and suppose the vertices $v_1, \ldots, v_n$ are in topologically sorted order. If there is a path from v_i to v_j in G, are we guaranteed that $i < j$?

(b) Let $G = (V, E)$ be any strongly connected directed graph. Is it always possible to remove one vertex (as well as edges to and from it) from the graph so that the remaining directed graph is strongly connected?

(c) Give an example where the greedy set cover algorithm does not produce an optimal solution.

(d) 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?

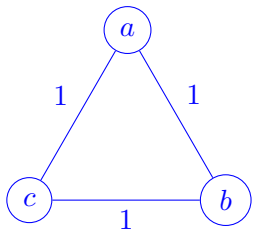
(e) The same question as above, except now no two edges can have the same length.

Solution:

(a) $\boxed{Yes}$ Consider a path $v_{i_1} \to \cdots \to v_{i_k}$. Since the graph is in topological order, we can conclude that $i_j < i_{j+1}$ for $j = 1, \ldots, k-1$, so by transitivity, $i_1 < i_k$.

(b) $\boxed{No}$ Consider the graph with 3 nodes a, b and c and edges $a \to b$, $b \to c$ and $c \to a$.

(c) With items $\boxed{B = \{a, b, c, d, e, f\}}$ and sets $\boxed{\{a, b, c\}, \{d, e, f\}, \{a, b, d, e\}}$ we would first choose {a,b,d,e} which would then require all three sets, while choosing the first two sets is sufficient.

(d) 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).

(e) 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.

3 (★★★) Arbitrage

Shortest-path algorithms can also be applied to currency trading. Suppose we have n currencies $C = \{c_1, c_2, \ldots, c_n\}$: e.g., dollars, Euros, bitcoins, dogecoins, etc. For any pair i, j of currencies, there is an exchange rate $r_{i,j}$: you can buy $r_{i,j}$ units of currency c_j at the price of one unit of currency c_i. Assume that $r_{i,i} = 1$ and $r_{i,j} \geq 0$ for all i, j.

(a) The Foreign Exchange Market Organization (FEMO) has hired Oski, a CS170 alumnus, to make sure that it is not possible to generate a profit through a cycle of exchanges; that is, for any currency $i \in C$, it is not possible to start with one unit of currency i, perform a series of exchanges, and end with more than one unit of currency i. (That is called *arbitrage*.) Give an efficient algorithm for the following problem: given a set of exchange rates $r_{i,j}$ and two specific currencies s, t, find the most advantageous sequence of currency exchanges for converting currency s into currency t. We recommend that you represent the currencies and rates by a graph whose edge lengths are real numbers.

[Provide 4 part solution.]

(b) In the economic downturn of 2016, the FEMO had to downsize and let Oski go, and the currencies are changing rapidly, unfettered and unregulated. As a responsible citizen and in light of what you saw in lecture this week, this makes you very concerned: it may now be possible to find currencies $c_{i_1}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \times r_{i_2,i_3} \times \cdots \times r_{i_{k-1},i_k} \times r_{i_k,i_1} > 1$. This means that by starting with one unit of currency c_{i_1} and then successively converting it to currencies $c_{i_2}, c_{i_3}, \ldots, c_{i_k}$ and finally back to c_{i_1}, you would end up with more than one unit of currency c_{i_1}. Such anomalies last only a fraction of a minute on the currency exchange, but they provide an opportunity for profit.

You decide to step up to help out the World Bank. Given an efficient algorithm for detecting the presence of such an anomaly. You may use the same graph representation as for part (a).

[Provide 4 part solution.]

Solution:

(a) **Main Idea:**
We represent the currencies as the vertex set V of a complete directed graph G and the exchange rates as the edges E in the graph. Finding the best exchange rate from s to t corresponds to finding the path with the largest product of exchange rates. To turn this

into a shortest path problem, we weigh the edges with the negative log of each exchange rate. Since edges can be negative, we use Bellman-Ford to help us find this shortest path.

Pseudocode:

1: **function** BESTCONVERSION(s, t)
2: $G \leftarrow$ Complete directed graph, c_i as vertices, edge lengths $l = \{-\log(r_{i,j}) \mid (i,j) \in E\}$.
3: $\texttt{dist}, \texttt{prev} \leftarrow$ BELLMANFORD(G, l, s)
4: **return** Best rate: $e^{-\texttt{dist}[t]}$, Conversion Path: Follow pointers from t to s in $\texttt{prev}$

Proof of Correctness:
To find the most advantageous ways to converts c_s into c_t, you need to find the path $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ maximizing the product $r_{i_1,i_2} r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k}$. This is equivalent to minimizing the sum $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}})$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. Because these weights can be negative, we apply the Bellman-Ford algorithm for shortest paths to the graph, taking s as origin. The correctness of the entire algorithm follows from the proof of correctness of Bellman-Ford.

Runtime:
Same as runtime of Bellman-Ford, $O(|V|^3)$ since the graph is complete.

(b) **Main Idea:**
Just iterate the updating procedure once more after $|V|$ rounds. If any distance is updated, a negative cycle is guaranteed to exist, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{k-1} r_{i_j,i_{j+1}} > 1$, as required.

Pseudocode: This algorithm takes in the same graph constructed in the previous part.

1: **function** HASARBITRAGE(G)
2: $\texttt{dist}, \texttt{prev} \leftarrow$ BELLMANFORD(G, l, s)
3: $\texttt{dist}^* \leftarrow$ Update all edges one more time
4: **return** True if for some v, $\texttt{dist}[v] > \texttt{dist}^*[v]$

Proof of Correctness:
Same as the proof for the modification of Bellman-Ford to find negative edges.

Runtime:
Same as Bellman-Ford, $O(|V|^3)$.

Note:
Both questions can be also solved with a variation of Bellman-Ford's algorithm that works for multiplication and maximizing instead of addition and minimizing.

4 (★★) Proof of Correctness for Greedy Algorithms

This question guides you through writing a proof of correctness for a greedy algorithm. A doctor's office has n customers, labeled $1, 2, \ldots, n$, waiting to be seen. They are all present right now and will wait until the doctor can see them. The doctor can see one customer at a time, and we can predict exactly how much time each customer will need with the doctor: customer i will take $t(i)$ minutes.

(a) We want to minimize the average waiting time (the average of the amount of time each customer waits before they are seen, not counting the time they spend with the doctor). What order should we use? You do not need to justify your answer for this part. (Hint: sort the customers by ____)

(b) Let $x_1, x_2, \ldots, x_n$ denote an ordering of the customers (so we see customer x_1 first, then customer x_2, and so on). Prove that the following modification, if applied to any order, will never increase the average waiting time:

- If $i < j$ and $t(x_i) \geq t(x_j)$, swap customer i with customer j.

(For example, if the order of customers is $3, 1, 4, 2$ and $t(3) \geq t(4)$, then applying this rule with $i = 1$ and $j = 3$ gives us the new order $4, 1, 3, 2$.)

(c) Let u be the ordering of customers you selected in part (a), and x be any other ordering. Prove that the average waiting time of u is no larger than the average waiting time of x—and therefore your answer in part (a) is optimal.

Hint: Let i be the smallest index such that $u_i \neq x_i$. Use what you learned in part (b). Then, use proof by induction (maybe backwards, in the order $i = n, n-1, n-2, \ldots, 1$, or in some other way).

Solution:

(a) Sort the customers by $t(i)$, starting with the smallest $t(i)$.

(b) First observe that swapping x_i and x_j does not affect the waiting time customers $x_1, x_2, \ldots, x_i$ or customers $x_{j+1}, x_{j+1}, \ldots, x_n$ (i.e., for customers x_k where $k \leq i$ or $k > j$). Therefore we only have to deal with customers $x_{i+1}, \ldots, x_j$, i.e., for customer k, where $i < k \leq j$. For customer x_k, the waiting time before the swap is

$$T_k = \sum_{1 \leq l < k} t(x_l),$$

and the waiting time after the swap is

$$T'_k = \sum_{1 \leq l < i} t(x_l) + t(x_j) + \sum_{i < l < k} t(x_l) = T_k - t(x_i) + t(x_j).$$

Since $t(x_i) \geq t(x_j)$, $T'_k \leq T_k$, so the waiting time is never increased for customers $x_{i+1}, \ldots, x_j$, hence the average waiting time for all the customers will not increase after the swap.

(c) Let u be the ordering in part (a), and x be any other ordering. Let i be the smallest index such that $u_i \neq x_i$. Let j be the index of x_i in u, i.e, $x_i = u_j$ and k be the index of u_i in x. It's easy to see that $j > i$. By the construction of x, we have $T(x_i) = T(u_j) \geq T(u_i) = T(x_k)$, therefore by swapping x_i and x_k, we will not increase the average waiting time. If we keep doing this, eventually we will transform x into u. Since we never increase the average waiting time throughout the process, u is the optimal ordering.

5 (★★★) 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 a linear-time algorithm that breaks up at least half the number of pairs of enemies as the best possible solution, and prove your answer.

You can treat the input as a graph $G = (V, E)$ with edge relations denoting enemies.

[Provide 4 Part Solution]

Solution:

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 $O(|V| + |E|)$.

6 (★★) 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 linear time algorithm for updating the tree. For each, use the four part algorithm format. However, for some of the cases the main idea, proof and justification of running time may be quite brief. Also, you may either write pseduocode for each case separately, or write a single algorithm which includes all cases.

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

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

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

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

Solution:

Main Idea

When working with MSTs, there are usually two things to try: look at cycles and look at cuts. When trying to decide if an edge will improve a spanning tree, we look at the cycle it forms when added to the MST. When trying to decide whether a spanning tree can be improved by removing an edge, we can look at the cut formed when we remove the edge from the tree. This is exactly what the algorithm below does.

Pseudocode

UPDATE MST($G = (V, E)$, $T = (V, E')$, w, e, $\hat{w}(e)$)
// Case 1
1. If $e \notin E'$ and $\hat{w}(e) > w(e)$: pass
// Case 2
2. If $e \notin E'$ and $\hat{w}(e) < w(e)$:
3. Add e to T and use BFS to find a cycle in T
4. Remove some heaviest edge in this cycle from T
// Case 3
5. If $e \in E'$ and $\hat{w}(e) < w(e)$: pass
// Case 4
6. If $e \in E'$ and $\hat{w}(e) > w(e)$:
7. Remove e from T creating two connected components of T, i.e. a cut of G
8. Use BFS to find every vertex on one side of this cut
9. Add the lightest edge crossing the cut to T

Proof of Correctness

Case 1: Note that increasing the weight of e either increases or doesn't affect every spanning tree of G. Since it doesn't affect T, it must still be an MST.

Case 2: First note that if an edge is not in an MST then it must be the heaviest edge in some cycle. Since no edges increased in weight, for every edge not in E' besides e, this is still the case. So we only have to decide if we need to add e to the tree, which is exactly what the algorithm does.

Case 3: Note that the weight of every spanning tree of G either stayed the same or decreased by $w(e) - \hat{w}(e)$. Since the weight of T decreased by $w(e) - \hat{w}(e)$ it is still an MST.

Case 4: First note that if some edge is in an MST then it must be the lightest edge across some cut (see discussion 5). Since no edges decreased in weight, for every edge in E' besides e this is still the case. So we only need to decide if we should remove e and if so, what edge should replace it. But we can only remove e if it is not the lightest edge across the cut examined by the algorithm, and if it is not then we must replace it by the lightest edge, which is just what the algorithm does.

Running Time First note that we can detect which case we are in simply by looping over the edges in T and checking if they are equal to e. Since T has $|V| - 1$ edges, this takes linear time.

Cases 1 and 3 take constant time since they do nothing. For case 2, it takes $O(|V|)$ time to run BFS on $T \cup \{e\}$ (since T has $|V| - 1$ edges) and find the heaviest edge in the cycle. In case 4, it takes $O(|V|)$ time to run BFS on $T - \{e\}$ in order to find the cut that only e (out of all the edges of T) crosses and then $O(|V| + |E|)$ time to find every edge crossing this cut and pick the lightest one (since in the worst case all but $|V| - 1$ edges in the graph cross the cut).

Thus the overall running time is $O(|V| + |E|)$. In fact, if we are not in case 4 then the running time is $O(|V|)$.