CS 170 HW 4

Due **2020-09-28, at 10:30 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer "Yes", "Yes but anonymously", or "No"

2 Counting Shortest Paths

This question is a solo question.

Given an undirected unweighted graph G and a vertex s, let $p(v)$ be the number of distinct shortest paths from s to v. We will use the convention that $p(s) = 1$ in this problem. Give an $O(|V| + |E|)$-time algorithm to compute $p(v)$ for all vertices. Only the main idea and runtime analysis are needed.

(Hint: For any v, how can we express $p(v)$ as a function of other $p(u)$?)

Solution: Main idea If v is distance $d > 0$ from s, the first $d-1$ edges in any shortest path to v will be a shortest path to a neighbor of v distance $d-1$ from s. Furthermore, this mapping from shortest paths to v and shortest paths to neighbors of v is bijective. So letting $N(v)$ be the set of neighbors of v at distance $d-1$, we get that $p(v) = \sum_{u \in N(v)} p(u)$.

Our algorithm is now: use BFS to compute all distances from s. Next, we set $p(s) = 1$, then for the remaining vertices in increasing distance order, we can compute $p(v) = \sum_{u \in N(v)} p(u)$. Since we look at vertices in increasing distance order, all u in $N(v)$ have $p(u)$ computed already.

Runtime analysis BFS takes $O(|V| + |E|)$ time. Computing $p(v)$ takes time $O(deg(v))$, so the total time to compute all $p(v)$ is $O(|E|)$.

3 Dijkstra Tiebreaking

This question is a solo question.

We are given a directed graph G with positive weights on its edges. We wish to find a shortest path from s to t, and, among all shortest paths, we want the one in which the longest edge is as short as possible. How would you modify Dijkstra's algorithm to this end? Just a description of your modification is needed.

(If there are multiple shortest paths where the longest edge is as short as possible, outputting any of them is fine).

Solution: Modify Dijkstra's algorithm to keep a map $\ell(v)$ which holds the longest edge weight on the current shortest path to v. Initially $\ell(s) := 0$ for source s and $\ell(v) := \infty$ for all $v \in V \setminus \{s\}$. When we consider a vertex u and its neighbour v, if $dist(u) + w(u,v) < dist(v)$, then we set $\ell(v) := \max(\ell(u), w(u,v))$ in addition to the standard Dijkstra's steps. If there

is a neighbour v of u such that $dist(v) = dist(u) + w(u,v)$, *and* $\ell(v) > \max(\ell(u), w(u,v))$, we set v's predecessor to u and update $\ell(v) := \max(\ell(u), w(u,v))$.

Note: An alternate solution that doesn't get full credit is, letting $W = \max_e w(e)$ and assuming all edge weights are integers, to add e.g. $n^{-1-(W-w(e)}$ to e's edge weight. Using these edge weights, of all the shortest paths, the one with the shortest possible longest edge will have the least weight, and no path can become shorter than a path it was previously longer than. However, writing down the edge weights with $n^{-1-(W-w(e)}$ added to them takes an additional $O(W \log n)$ bits per edge weight. As we saw in discussion, algorithms with runtime polynomial in the weights of edges are **not** efficient in general.

4 Bounded Bellman-Ford

Modify the Bellman-Ford algorithm to find the weight of the lowest-weight path from s to t with the restriction that the path must have at most k edges. Just a description of your modification is needed.

Solution: The obvious instinct is to run the outer loop of Bellman-Ford for k iterations instead of $|V|-1$ iterations. This fails because it's possible that we find a lowest-weight path from s to some vertex using $|V|-1$ edges in the first iteration, depending on the order in which we iterate over edges (e.g., see Discussion 4, Problem 1b). Intuitively, to avoid this we want to use the "old" values of $dist$ in each iteration.

So, let $dist_i(v)$ be the lowest-weight of a path from s to t using at most i edges. $dist_0(s) = 0$, and all other $dist_i(v)$ are initialized to ∞. Instead of doing the update $dist(v) = min(dist(v), dist(u) + w_{u,v})$, in iteration i of Bellman-Ford we do the update $dist_i(v) = min(dist_i(v), dist_{i-1}(u) + w_{u,v})$. We run for k iterations, then output the values $dist_k$.

5 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 of currencies c_i, c_j, 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.

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*.)

More precisely, arbitrage is possible when there is a sequence of currencies $c_{i_1}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$. 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.

We say that a set of exchange rates is arbitrage-free when there is no such sequence, i.e. it is not possible to profit by a series of exchanges.

(a) Give an efficient algorithm for the following problem: given a set of exchange rates $r_{i,j}$ which is *arbitrage-free*, and two specific currencies s, t, find the most profitable sequence

of currency exchanges for converting currency s into currency t. That is, if you have a fixed amount of currency s, output a sequence of exchanges that gets you the maximum amount of currency t.

Hint: represent the currencies and rates by a graph whose edge weights are real numbers.

(b) Oski is fed up of manually checking exchange rates, and has asked you for help to write a computer program to do his job for him. Give an efficient algorithm for detecting the possibility of arbitrage. You may use the same graph representation as for part (a).

For both parts, give a three-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, i.e. set $w_{ij} = -\log r_{ij}$. Since edges can be negative, we use Bellman-Ford to help us find this shortest path.

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 $w_{ij} = -\log r_{ij}$. 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.

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.

6 Updating a MST

This question is a solo question.

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 using both w and $\hat{w}$.

(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:

(a) **Main Idea:** Do nothing.

Correctness: T's weight does not increase, and any other spanning tree's weight either stays the same or increases, so T must still be the MST.

Runtime: Doing nothing takes $O(1)$ time.

(b) **Main Idea:** Add e to T. Use DFS to find the cycle that now exists in T. Remove the heaviest edge in the cycle from T.

Correctness: The heaviest edge in a cycle cannot be in the MST (because if it is in the MST, you can remove it from the MST and add some other edge to the MST, and the MST's cost will decrease), and any edge not in an MST is the heavest edge in some cycle (in particular, the cycle formed by adding it to the MST). For any edge not in T except for e, decreasing e's weight does not change that it is the heaviest edge in the cycle, so it is safe to exclude from the MST. By adding e to T and then removing the heaviest edge in the cycle in T, we remove an edge that is also not in the MST. Thus after this update, all edges outside of T cannot be in the MST, so T is the MST.

Runtime: This takes $O(|V|)$ time since T has $|V|$ edges after adding e, so the DFS runs in $O(|V|)$ time.

(c) **Main Idea:** Do nothing.

Correctness: T's weight decreases by $w(e) - \hat{w}(e)$, and any other spanning tree's weight either stays the same or also decreases by this much, so T must still be an MST.

Runtime: Doing nothing takes $O(1)$ time.

(d) **Main Idea:** Delete e from T. Now T has two components, A and B. Find the lightest edge with one endpoint in each of A and B, and add this edge to T.

Correctness: Every edge besides e in the MST is the lightest edge in some cut prior to changing e's weight, and increasing e's weight cannot affect this property. So all edges besides e are safe to keep in the MST. Then, whatever edge we add is also the lightest edge in the cut (A, B) and is thus also in the MST.

Runtime: This takes $O(|V| + |E|)$ time, since it might be the case that almost all edges in the graph might have one endpoint in both A and B and thus almost all edges will be looked at.

7 Job Scheduling

There are n computing jobs to be run on two supercomputers. Each supercomputer can only run one job at a time, and the jth job has processing time $t_j \geq 0$. We want to come up with a *schedule* for these jobs. That is, we want to assign these n jobs to one of the two supercomputers, and decide in what order each supercomputer will run the jobs assigned to it.

For each job i let J_i be the set of jobs run before job i on the supercomputer job i is assigned to. Then job i has *waiting time* $\sum_{j \in J_i} t_j$. Our goal is to minimize the total waiting time of all jobs, $\sum_{i \in [n]} \sum_{j \in J_i} t_j$.

For example, if the jobs have processing times 1, 2, 3, 4, 5, and we assign the first supercomputer the jobs with processing times 1, 3, 2 in that order, and the second supercomputer the jobs with processing times 4, 5 in that order, the total waiting time is $0 + 1 + 4 = 5$ for jobs on the first supercomputer and $0 + 4 = 4$ for jobs on the second supercomputer, so the total waiting time is 9.

(a) Suppose each supercomputer must process exactly $n/2$ jobs, assuming n is even. Given the processing times t_j for each job, give an efficient algorithm for assigning jobs to each supercomputer and ordering the jobs assigned to each.

(Hint: Recall the service scheduling problem from discussion)

(b) Now, you can assign any number of jobs to each supercomputer. Show how we can solve this problem by constructing an instance of part (a) that has the same optimal cost.

Just a main idea and proof of correctness are needed for both parts.

Solution:

(a) **Main idea:** We can use a greedy strategy, adding jobs in order of increasing size and alternating between the supercomputers.

Proof of correctness:

Let $t_{i,1}, t_{i,2}$ be the length of the ith jobs run by the first and second supercomputer respectively. The total waiting time of a solution can be expressed as

$$\sum_{i=1}^{n/2} (n/2 - i) * (t_{i,1} + t_{i,2})$$

since the ith job run by each supercomputer contributes t_j waiting time to the $n/2 - i$ jobs run after it. Note that swapping the ith jobs in the two lanes does not increase the cost, and swapping a shorter job with a longer job in front of it can only decrease the cost. We can arrive at our solution from any other solution only using these swaps, so our solution must be optimal.

(b) There are two acceptable solutions: Solution 1 adds to the proof of part (a) by showing that the optimal solution is always "balanced". Solution 2 creates a new set of jobs such that the optimal cost in part (a) for the new set of jobs is equal to the cost in part (b) of the original set of jobs.

Solution 1:

To add to the proof of correctness in part (a), consider any solution where the number of jobs on the two supercomputers differs by more than 1. We can take the first job i on the supercomputer with more jobs, and move it to be the first job on the supercomputer with less jobs. This can only decrease the cost of the solution, since this job goes from contributing at least $\frac{n-1}{2}t_i$ to the total waiting time to at most $\frac{n-1}{2}t_i$. With this and the swaps described in part (a), we can transform any solution into the solution we described in part (a).

Solution 2:

Main idea: We add n dummy jobs, all with $t_j = 0$, and restrict each supercomputer to running n jobs to get an instance of part (a).

Proof of correctness: It suffices to show that for any solution to the part (a) instance, there is a solution to part (b) of the same or lesser cost and vice-versa. This implies the two problems have the same optimal cost.

Given any part (b) solution, we can get a solution to the part (a) instance by "padding" each of the lanes with the n dummy jobs in front until each supercomputer has n jobs in it. This does not increase the cost.

Given any solution to the part (a) instance, we can delete all dummy jobs to get a valid solution to part (b), and the cost can only decrease in doing this.

8 (Extra Credit) Min-Sum Multiplication

In the all-pairs shortest path (APSP) problem, we are given an undirected connected graph G with positive edge weights. We want to find the length of the shortest path between *every* pair of vertices.

Closely related to shortest path problems is the min-sum product. The dot product of two vectors a, b is $\sum_i a_i \cdot b_i$. The min-sum product of two vectors is instead $\min_i(a_i + b_i)$. The min-sum product of two matrices is defined analogously to the normal product of two matrices - for two n-by-n matrices A, B, their min-sum product's (i, j)-th entry is the min-sum product of the ith row of A and the jth column of B (instead of their dot product).

Suppose we can compute the min-sum product of two n-by-n matrices in $T(n)$ time. Give an efficient algorithm for the APSP problem, and express your runtime in terms of $T(n)$.

Assume $n^2 = O(T(n))$, i.e. the time to write down an n-by-n matrix is not more than the time to multiply two of them. **Give a three-part solution**.

Solution: Main idea: Let us treat G as having self-loops on all vertices of length 0.

Let A be the matrix where $A_{i,j} = w_{ij}$ if (i, j) is an edge in the graph, and $A_{i,j} = \infty$ if (i, j) is not an edge in the graph.

We compute A^n by doing $\log n$ min-sum squarings, i.e. $A^2 = A * A, A^4 = A^2 * A^2, \ldots$. For all i, j pairs, we output $A^n_{i,j}$ as the shortest path length between i, j.

(If we don't want entries with value ∞, our algorithm will work if we just replace ∞ with a number guaranteed to be larger than any shortest path length in the graph, e.g. $n^{100} * \max_e w_e$.)

Proof of correctness: If we allow repeat edges, the shortest path between any two vertices is identical to the shortest path between any two vertices of length n, since we can pad any shortest path with the self-loops in G. We will show our algorithm computes the shortest path between any two vertices of length n.

Consider the entries of A^2. $A^2_{i,j} = \min_k[A_{i,k} + A_{k,j}]$. Note that this is ∞ if one of $(i, k), (k, j)$ is not an edge in the graph, and otherwise, it's the length of a 2-edge path from i to j. Furthermore, the minimization is over all possible length-2 paths. So $A^2_{i,j}$ is the length of the shortest path from i to j of length 2.

By a similar argument $A^\ell_{i,j} = \min_{k_1,k_2,\ldots k_{\ell-1}}[A_{i,k_1} + A_{k_1,k_2} + \ldots + A_{k_{\ell-1},j}]$, i.e. the shortest path from i to j of length ℓ.

Runtime:

We do $\log n$ min-sum matrix multiplications, so the runtime is $O(T(n) \log n)$.