CS 170 Homework 5 (Optional)

Due **10/2/2023, at 10:00 pm (grace period until 11:59pm)**

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".

2 Counting Shortest Paths

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 $p(v) \bmod 1337$ for all vertices. Only the main idea and runtime analysis are needed.

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

Note: As a secondary question, you should ask yourself whether the runtime would remain the same if we were computing $p(v)$ rather than $p(v) \bmod 1337$.

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 $p(v) \bmod 1337 = \sum_{u \in N(v)} p(u) \bmod 1337$. 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 $p(v) \bmod 1337$ from its neighbors takes time $O(deg(v))$, so the total time to compute all $p(v) \bmod 1337$ is $O(|E|)$.

If we did not have the $\bmod 1337$, and we instead wanted $p(v)$ exactly, we would have a higher runtime, as the number of length-d paths to v can be exponential in d and so arithmetic on these numbers would not be constant time.

3 The Greatest Roads in America

Arguably, one of the best things to do in America is to take a great American road trip. And in America there are some amazing roads to drive on (think Pacific Coast Highway, Route 66 etc). An intrepid traveler has chosen to set course across America in search of some amazing driving. What is the length of the shortest path that hits at least k of these amazing roads?

Assume that the roads in America can be expressed as a directed weighted graph $G = (V, E, d)$, and that our traveler wishes to drive across at least k roads from the subset $R \subseteq E$

of "amazing" roads. Furthermore, assume that the traveler starts and ends at her home $a \in V$. You may also assume that the traveler is fine with repeating roads from R, i.e. the k roads chosen from R need not be unique.

Design an efficient algorithm to solve this problem. Provide a 3-part solution with runtime in terms of $n = |V|$, $m = |E|$, k.

Hint 1: Create a new graph G' based on G such that for some s', t' in G', each path from s' to t' in G' corresponds to a path of the same length from a to itself in G containing at least k roads in R. It may be easier to start by trying to solve the problem for $k = 1$.

Solution: Main idea:
We want to build a new graph G' such that we can apply Dijkstra's algorithm on G' to solve the problem.

We'll start by creating $k + 1$ copies of G. Call these $G_0, G_1, \ldots G_k$. These copies include all the edges and vertices in G, as well as the same weights on edges. Let the copy of v in G_i be denoted by v_i. For each road $(u, v) \in R$, we also add the edges $(u_0, v_1), (u_1, v_2), \ldots (u_{k-1}, v_k)$, with the same weight as (u, v). The intuition behind creating these copies is that each time we use an edge in G' corresponding to an edge in R, we can advance from one copy of G to the next, and this is the only way to advance to the next copy. So if we've reached v_i from a_0, we know we must have used (at least) i edges in R so far.

Now, consider any path p' from a_0 to a_k in G'. If we take each edge (u_i, v_i) or (u_i, v_{i+1}) and replace it with the corresponding edge (u, v) in G, we get a path p in G from a to itself. Furthermore, since the path goes from a_0 to a_k, it contains k edges of the form (u_i, v_{i+1}), where (u, v) is an edge in R. So, p will contain at least k edges in R.

Our algorithm is now just to create G' as described above, and find the shortest path p' from a_0 to a_k using Dijkstra's, and then output the corresponding path p in G.

Correctness:
Assume there is a valid path p in G that is shorter than the one produced by this algorithm. Consider the equivalent path p' in G' formed by modifying the path to go to the next copy of G whenever an edge of R is crossed. Since p is valid, p' must go from a in G_0 to a in G_k. But then p' would be a shorter path in G' than the one produced by Dijkstra's, which is a contradiction.

Runtime:
Since G' includes $k+1$ copies of G, Dijkstra's algorithm will run in time $O((km+kn)\log(kn))$.

4 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 $(r_{i,j})_{i,j \in n}$ which is *arbitrage-free*, and two specific currencies a, b, find the most profitable sequence of currency exchanges for converting currency a into currency b. That is, if you have a fixed amount of currency a, output a sequence of exchanges that gets you the maximum amount of currency b.

Hint 1: represent the currencies and rates by a graph whose edge weights are real numbers.

Hint 2: $\log(xy) = \log(x) + \log(y)$

(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 (a) and (b), 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 a to b 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_a into c_b, 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.

5 Sum of Products

This question guides you through writing a proof of correctness for a greedy algorithm. You have n computing jobs to perform, with job i requiring t_i units of CPU time to complete. You also have access to n machines that you can assign these jobs to. Since the machines are in high demand, you can only assign one job to any machine. The jth machine costs c_j dollars for each unit of CPU time it spends running a job, so assigning job i to machine j will cost you $t_i \cdot c_j$ dollars (each job takes the same amount of CPU time to complete, regardless of which machine is used). Your goal is to find an assignment of jobs to machines that minimizes the total cost.

Assume the jobs and machines are sorted and have distinct runtimes/costs, i.e. $t_1 > t_2 > \ldots > t_n$ and $c_1 > c_2 > \ldots > c_n$.

(a) Describe the assignment of jobs to machines minimizes the total cost (no proof necessary).

Hint: What machine should we assign the longest job to? What machine should we assign the second longest job to? It might help to solve a small example by hand first.

(b) Given an assignment of jobs to machines, consider the following modification: If there is a pair of jobs i, j such that job i is assigned to machine i', job j is assigned to machine j', and $t_i > t_j$ and $c_{i'} > c_{j'}$, instead assign job i to machine j' and job j to machine i'. Show that this modification decreases the total cost of an assignment.

(c) Use part b to show that the assignment you chose in part a has the minimum total cost (Hint: Show that for any assignment other than the one you chose in part a, you can apply the modification in part b. Conclude that the assignment you chose in part a is the optimal assignment.)

Solution:

(a) Assign the longest job (job 1) to the cheapest machine (machine n), the second longest job (job 2) to the second cheapest machine (machine $n-1$), etc.

(b) The cost of computing jobs other than i and j is unaffected. The old cost of computing jobs i and j is $t_i c_{i'} + t_j c_{j'}$. The new cost of computing jobs i and j is $t_i c_{j'} + t_j c_{i'}$. The decrease in cost is then $t_i c_{i'} + t_j c_{j'} - t_i c_{j'} - t_j c_{i'} = (t_i - t_j)(c_{i'} - c_{j'})$, which is positive because $t_i > t_j$ and $c_{i'} > c_{j'}$.

(c) Suppose for an assignment of jobs to machines that no modification of the type suggested in part b is possible. Then job 1 must be assigned to machine n in this assignment - otherwise job 1 and whatever job is assigned to machine n satisfy the conditions in part b. But if we know job 1 is assigned to job n, then we can similarly conclude that job 2 must be assigned to machine $n-1$, and so on. So this assignment must be exactly the assignment we chose in part a.

Then for every assignment except the assignment we found in part a, there is another assignment that has lower total cost, i.e. the former is not optimal. This implies that the assignment from part a is the optimal assignment.

6 Rigged Tournament

Peter is in charge of organizing a football tournament with n teams. The tournament is a single-elimination tournament: if teams i and j play, the team that loses is out of the tournament and cannot play any more games. There are no ties.

Peter's shady friend Jeff has given him the following inside information: if teams i and j play, then they will score a combined total of $f(i, j) \geq 0$ points in that game and furthermore Jeff can rig the match so that the team of his choice wins. Peter wishes to find a tournament schedule which (1) maximizes the number of points scored in the tournament, and (2) makes his favorite team, team i^*, win the tournament. Give an efficient algorithm to solve this problem and provide its runtime; proof of correctness is not required.

Note: teams need not play an equal number of games in the tournament. For example, if the teams are $\{1, 2, 3, 4\}$, then a valid tournament schedule (where (i, j) means i plays j and i wins) is $[(1, 2), (1, 3), (4, 1)]$. Here, team 4 wins the tournament.

Solution: Let $G = (V, E)$ be the complete graph on n vertices with edge weights $\ell(i, j) = f(i, j)$. The key observation is that a tournament schedule corresponds to a tree in this graph, and the weight of the tree is the number of points scored across all games in that tournament schedule. Therefore, let T be the Maximum Spanning Tree in G. The weight of T is the maximum number of points scored in the tournament. To extract the sequence of games, simply run DFS or BFS from i^*, and play the games by largest depth first, making the team with a lower depth always win. For the runtime, DFS/BFS takes O(n) time, which means the MST algorithm is the bottleneck. This takes $O(|E| \log |V|) = O(n^2 \log n)$ since there are $O(n^2)$ edges (the complete graph).

7 Twenty Questions

Your friend challenges you to a variant of the guessing game 20 questions. First, they pick some word $(w_1, w_2, ..., w_n)$ according to a known probability distribution $(p_1, p_2, ..., p_n)$, i.e. word w_i is chosen with probability p_i. Then, you ask yes/no questions until you are certain which word has been chosen. You can ask any yes/no question, meaning you can eliminate any subset S of the possible words with the question "Is the word in S?".

Define the cost of a guessing strategy as the expected number of queries it requires to determine the chosen word, and let an optimal strategy be one which minimizes cost. Design an $O(n \log n)$ algorithm to determine the cost of the optimal strategy.

Give a 3-part solution.

Note: *We are only considering deterministic guessing strategies in this question. Including randomized strategies doesn't change the answer, but it makes the proof of correctness more difficult.*

Solution:

This solution is inspired by the observation that in binary coding, each bit of a codeword we read further narrows the possible symbols being encoded, just like a question in the game above. This correspondence is made rigorous in the proof of correctness.

Main idea: Create a Huffman tree on $(w_{1\cdots n})$ with weights $(p_{1\cdots n})$ and return the expected length of a codeword under the corresponding encoding.

Proof of correctness: Note that any guessing strategy gives a prefix-free binary encoding of the words $(w_{1\cdots n})$, where each word w_i is encoded by sequences of yes/no answers which would lead you to conclude that w_i was chosen. This encoding is prefix-free because the game only ends when all words except one have been eliminated.

Additionally, any prefix-free encoding of the words can be made into a guessing strategy as follows. Let $x_n \in \{0,1\}^n$ represent the sequence of yes/no answers received on the first n questions, with 1 corresponding to yes and 0 corresponding to no. Then asking at step $n+1$ whether the word can be encoded by a string with the prefix $x_n \circ 1$ (i.e. the answers so far followed by a yes) will result in the final sequence of answers being a valid encoding of the chosen word.

In this correspondence, the expected code length equals the cost of a guessing strategy. Therefore finding an optimal strategy is equivalent to finding a prefix-free encoding of the words with minimum expected codelength, which is exactly what Huffman coding does. To get the final answer, we calculate the expected codelength of the optimal strategy, which can be done by DFS from the root of the Huffman tree.

Runtime: A Huffman tree can be constructed in $n \log(n)$ time (this is dominated by the time to sort the probabilities). The average codelength (and thus cost of the associated strategy) can be calculated in $O(n)$ time by DFS. Therefore the total runtime is $O(n \log(n))$.

8 Preventing Conflict

A group of n guests shows up to a house for a party, but the host knows that m pairs of these guests are enemies (a guest can be enemies with multiple other guests). 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 $O(n+m)$-time algorithm that breaks up at least half of the pairs of enemies. **Give a three-part solution.**

Solution:

Main Idea: We add the guests one-by-one, each time adding a guest to the room that currently has less of their enemies (in the case of a tie, we choose one of the rooms arbitrarily).

Proof of Correctness: Let m_i be the number of the ith guest's enemies that have already been added to a room when we're adding the ith guest. We break up at least $m_i/2$ of these enemy pairs by adding this guest to the room with less of their enemies. So we break up at least $\sum_i m_i/2$ of the $\sum_i m_i$ total enemy pairs as desired.

Running Time: Each guest is iterated through once, and we can figure out which room to assign them in time proportional to the number of enemies they have (using e.g. an array tracking which room we've assigned each individual to so far), i.e. each pair of enemies is considered at most twice, so the total time is $O(n+m)$.

(Comment: This problem is equivalent to the max-cut problem: Given an undirected graph, in the max cut problem we want to split the vertices into two sets so that as many edges as possible have one endpoint in each set)