CS 170: Efficient Algorithms and Intractable Problems

Spring 2017

•

Homework 4

Due on Tuesday, Februrary 28h, 2017 at 11:59am

•

Solutions by

Michael Fan

26697596

In collaboration with

None

Problem 1: Short Questions

★★ **Level**

For the following claims, answer yes or no and provide justification. Consider an arbitrary graph G with positive edge weights. Shortest path means least cost path.

a. Let G be a connected undirected graph with positive length 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?

> **Solution** No. Consider graph G with vertices $\{A, B, C, D\}$ and edges $\{(A, B), (B, C), (C, D), (D, A)\}$ all of length 2. Let A be our fixed vertex. Then B is a vertex that minimizes $d(A, B)$. One possible shortest path from A to B is (A, B) with length 2, but (A, B) is not part of the MST $(B, C), (C, D), (D, A)$.

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

> **Solution** Yes. We know that s and v have to be adjacent. If they are not, then there exists a vertex v' on the minimal distance path from s to v that is even closer to s which is a contradiction. Now, let us make a cut between s and the rest of G. We know that v is the vertex which minimizes $d(s, v)$ for any vertex in G. Therefore, by the cut principle, (s, v) must be included in any MST as it is the shortest edge (no chance for a tie because all edge lengths are unique) across the cut, $(s, G - s)$.

Problem 2: Huffman Encoding

★★ **Level**

We use Huffman's algorithm to obtain an encoding of alphabet a, b, c with frequencies f_a, f_b, f_c. In each of the following cases, either give an example of the frequencies (f_a, f_b, f_c) that would yield the specified code, or explain why the code cannot possibly be obtained (no matter what the frequencies are)¿

a. Code: $\{0, 10, 11\}$

Solution This code sequence can be achieved with the frequencies, $\{f_a, f_b, f_c\} = \{3, 2, 1\}$

b. Code: $\{0, 1, 00\}$

Solution This is impossible because 0 and 00 share the same prefix.

c. Code: $\{10, 01, 00\}$

Solution This is impossible because this encoding is not optimal. Following the algorithm for Huffman encoding, an alphabet of 3 elements can be encoded optimally with 2 binary strings of 2 bits (the least frequent 2 elements) and 1 with 1 bit that is the root node.

Problem 3: Preventing Conflict

★★★★ Level

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.
Hint: Try assigning guests one at a time. Consider how many pairs of enemies are broken up with each iteration.

Solution
Overview:
For every next person that comes to the party, assign him to the room where he has fewer enemies.

Pseudocode:
function splitEnemies(G):
//Input: G(V, E) in the form of 2 adjacency lists - one for friends and one for enemies
//Output: All v ∈ V are split between set 1 or set 2
    set1 = []
    set2 = []
    for v in V:
        enemiesIn1 = 0, enemiesIn2 = 0
        for person in set1:
            if person in v's enemy list, increment enemiesIn1
        for person in set2:
            if person in v's enemy list, increment enemiesIn2
        if enemiesIn1 ≥ enemies2, put v in set2, else put v in set1

Proof of Correctness:
This problem can be modeled with a maximally connected graph G with n vertices where each vertex represents a different guest. Each edge represents the relationship between 2 people. The edges are encoded in the adjacency lists of the vertices - if 2 vertices have each other in their friends lists or enemies lists, then they are friends or enemies intuitively. The worst case scenario is when everyone hates everyone else. In this case, the best that the optimal algorithm can do is to split everyone into 2 equal groups, in which $\frac{|V|^2}{4}$ enemies relations are broken, which is equal to the number of edges in a bipartite graph. To prove that this is the best that any algorithm can do: if any one person is shifted to the other room in this scenario, $\frac{|V|}{2} - 1$ relationships are severed from the room that the person came from but $\frac{|V|}{2}$ are added in the room that the person is moved to which results in a net decrease of severed relationships. Our algorithm can do just that. We always place the next person in the room with the fewest people that she hates, so the number of hate relationships in each room is always minimized. We will proceed via induction to show that the claimed severed number of relationships is correct.
Base case of 1 person: we place that person in either room and we are done. 0 relationships are severed: $0 = floor(\frac{1^2}{4})$.
Base case of 2 people who are enemies: we put one in room 1 and one in room 2 and 1 enemy relationship is severed: $1 = \frac{2^2}{4}$.
Inductive hypothesis: assume that with $2n$ people who hate each other, we sever $\frac{(2n)^2}{4} = n^2$ relationships by placing n people in each room.
Inductive step: assume that now there are $2n+2$ people who hate each other. We take 2 more people instead of 1 so that there is no need for flooring. Each added person contributes an addition number of severed relationships $= n+1$ as they are isolated from the $n+1$ people in the other room. But, we have overcounted by 1 because there is only one relationship between the 2 new added people. So, the total number of severed relationships is given by $n^2 + 2n + 1 = (n + 1^2)$ and we conclude our proof.

Solution (cont.)

Runtime Analysis: For each new person, we check through the enemy adjacency lists of everyone in both rooms that came before them. It takes constant time to check if 2 people are friends or enemies as the graph is given beforehand. Assignment to a room is constant time. Thus, for each new person, it takes $O(n * 1) = O(n)$ to figure out which set they should be added to and $O(1)$ to push them into a set. For n people, this algorithm then takes $O(n^2)$.

Problem 4: Graph Subsets

★★ **Level**

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?

   Solution No. The optimal solution will have stray negative edges because they always decrease the overall weight of the graph.

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

   Solution Kruskal's algorithm will not solve this problem. Take for example graph G with V = $\{A, B, C\}$ and E = $\{(A, B), (B, C), (C, A)\}$ all with weight -1. Then, Kruskal's algorithm will output either $\{(A, B), (B, C)\}$, $\{(B, C), (C, A)\}$, or $\{(C, A), (A, B)\}$. However, if we add in the missing edge to each of these MSTs, we end with a solution with weight -3 instead of -2. Thus, Kruskal's algorithm does not solve this problem.

3. Briefly describe an efficient algorithm for this problem. Just the main idea is enough (1-3 sentences). No need for a 4-part solution.

   Solution An efficient algorithm for this problem would consist of first running Kruskal's algorithm and them adding all negative edges in the graph to the set. This would ensure that all vertices in the graph are indeed spanned with an MST, but also that any negative optimizations are included in E'.

Problem 5: Arbitrage

★★★ **Level**

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 $r_{i,j}$ unites of currency c_j at the price of one unit of currency c_i. Assume that $r_{i,j} = 1$ 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, 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.

Solution

Overview:
Set a graph with the vertices set to infinity except for s which is initialized to 0, and the edge lengths set to negative log of the conversion rates. Then, run Bellman-Ford.

Pseudocode:

```
function bestRate(map_conversions, s, t):
//Input: map of conversions where the keys are tuples of currencies (a, b) and the value is the conversion rate from a to b, s is the starting currency, t is the currency that we want to convert to from s
//Output: the best conversion path of one currency to another
    create empty graph G
    for key in map_conversions:
        if a or b in key are not vertices of G, add them as vertices of G
        add directed edge from a to b with weight = -log(map_conversions.get(key))
    for v in V:
        dist[v] = infinity
    dist[s] = 0
    bellman_ford(G, s, t) with |V|-1 cycles
    return path ending at t starting from s
```

Proof of Correctness:
We want to find the best conversion sequence from s to t. By encoding the negative logs of the conversion rates as the path lengths of our graph and then running Bellman-Ford, we ensure that we find the "shortest path" from s to t which is in fact the longest path in terms of the real conversion factor. In doing this, we take advantage of the log property, $log(a) + log(b) = log(ab)$. Thus, when we add path lengths, we are in fact multiply together the exchange rates and maximizing them as we should be. Bellman-Ford is proven to be able to find the shortest path (which in our case is really calculating the largest exchange factor as explained previously) and thus by setting the edge lengths to negative logs of the conversion rates, we find the string of conversion rates that multiply to the greatest product possible.

Runtime Analysis:
This algorithm consists of creating a graph G and then running Bellman-Ford. Creating a graph takes $O(|V| + |E|)$ and Bellman-Ford takes $O(|V||E|)$ time. Thus, this algorithm is dominated by Bellman-Ford and takes $O(|V||E|)$ time.

(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 many now be possible to find currencies $c_{i_1}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \times r_{i_2,i_3} \times \ldots \times r_{i_{k-1},i_k} \times r_{i_k,i_1} > 1$. 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 and opportunity for profit.
You decide to step up and 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).

Solution
Overview:
Run Bellman-Ford $|V|$ times on the same graph G that was constructed in part (a). If there are any updates on the last iteration, then there exists a negative cycle in the graph, meaning that there is arbitrage.

Pseudocode:
function detectArbitrage(map_conversions):
//Input: map of conversions where the keys are tuples of currencies (a, b) and the value is the conversion rate from a to b
//Output: if there is potential arbitrage in the conversion rates
    create empty graph G
    for key in map_conversions:
        if a or b in key are not vertices of G, add them as vertices of G
        add directed edge from a to b with weight = -log(map_conversions.get(key))
    for v in V:
        dist[v] = infinity
    dist[s] = 0
    bellman_ford(G, s, t) with $|V|$ cycles
    return true if there was an update to the weights in the last update, otherwise return false

Proof of Correctness:
Bellman-Ford can detect negative cycles: if there is an update to any weight on the $|V|$th iteration of the algorithm, then there is a negative cycle in the graph. A negative cycle in our constructed graph means that there is a series of conversions for which the product is greater than 1. Thus, the algorithm is correct in detected any instances of arbitrage.

Runtime Analysis:
Identical to part (a): we construct the graph and then run Bellman-Ford one additional time than we did in part (a) which makes no difference in asymptotic runtime. Thus, the runtime is still $|V||E|$.