CS 170: Efficient Algorithms and Intractable Problems

Spring 2017

•

Homework 4

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

•

Solutions by

Aiswarya Sankar

26347114

In collaboration with

Jacob Holesinger, Sudeep Dasari

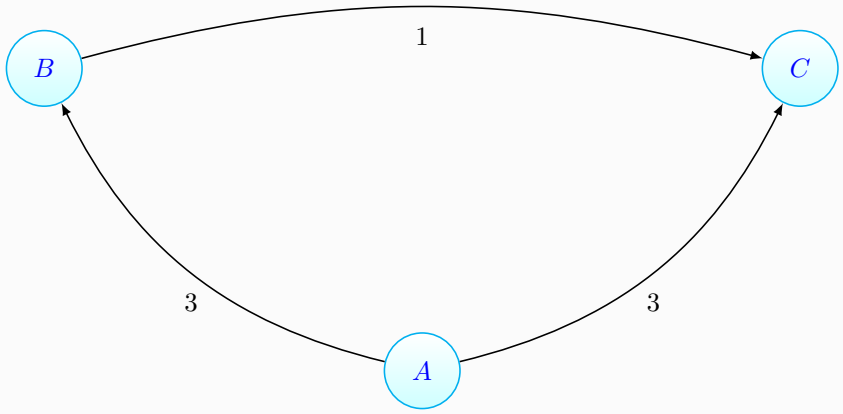
Problem 1: Short Questions

★★ 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 We can disprove this by counterexample. For example given the graph with the following edges and vertices we have



Given this graph we can show that a valid MST would include edges BC and AB or AC. It wouldn't include both AC and AB because that would not fit the invariant that a minimum spanning tree minimizes the total weight of the graph's edges. Thus every edge on the path from s to v isn't a part of every minimum spanning tree of G and we have provided a counterexample.

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

Solution This is true. We know that since every vertex in the graph must be connected we can show that the minimum path leaving a given vertex must be included in the MST if all the edge weights are unique.
Proof by contradiction: Assume that the every edge on the path from s to v isn't a part of every MST of G. This means that out of a vertex there is an edge E' that is chosen over E where E' ¿ E. This however invalidates the MST invariant that the total sum of the edge weights must be minimized. Therefore by contradiction we have shown that every edge on the path from s to v must be a part of every minimum spanning tree of G.

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 can be obtainded if $f_a > f_b \geq f_c$. Therefore we can assign $f_a = 0.5, f_b = .25, f_c = .25$.

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

> **Solution** This code isn't a valid Huffman encoding since 0 is a prefix of 00 and Huffman encoding needs to guarantee that all codes are prefix free.

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

> **Solution** This is also not a valid encoding for two reasons. First we can show that this solution isn't a full binary tree. In order to be a full binary tree each node must have either 0 or 2 children. In this example however node 1 has 1 child 10. The reason this is invalid is that we could instead use the encoding 1, 01, 00 which would also be a valid encoding and give us a smaller overall sum of weights.

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

Main Idea

We want to assign each person to a room one at a time. Thus intuitively we ask the question: given a set A of people in room 1 and set B of people in room 2, which room would we rather put person x in? We want to put person x in the room with fewer enemies because this maximizes the number of enemy ties broken with that placement. Thus the question becomes, how do we efficiently compute the number of enemies a person has in each room? We know that sets implemented using hashtable has constant lookup time. Thus as we place each person in a room we will store the set of people in that room in a hashset.
Our graph is represented with nodes as people and edges as enemy ties. Thus for each person we are going to place in a room, we iterate through the person's edges and place the person in the corresponding room.

Psuedocode

```
def separateEnemies(enemies):
   set1 = {}
   set2 = {}
   for person in enemies:
      enemiesSet1, enemiesSet2 = 0
      for enemy in person.neighbors:
         if enemy in set1:
            enemiesSet1 += 1
         if enemy in set2:
            enemiesSet2 += 1
         else:
            add(person) to smaller set
      if enemiesInSet1 >= enemiesInSet2:
         set1.add(person)
      else:
         set2.add(person)
```

Proof of Correctness

We need to prove that this algorithm will break up at least half the number of pairs of enemies. We can prove this because for each person we are placing him or her in the set in which he or she has fewer enemies. Assume a person x has n enemies. Then there are the following conditions to consider:

$$\begin{cases} enemiesRoom1 > n/2: & \text{break} > \text{n/2 enemies} \\ enemiesRoom1 == n/2: & \text{break n/2 enemies} \\ enemiesRoom1 < n/2: & \text{break} > \text{n/2 enemies} \end{cases}$$

This way we are guaranteed to break up at least n/2 enemy pairs each time where n is the total number of enemies a person has. Therefore since we do this for each person we guarantee that as a sum we will break n/2 enemy pairs.

Solution (cont.)

Runtime Analysis

This will be $O(E)$ where E is the number of edges. We will represent this problem as a graph where edges are enemy relationships and the people in each room is a hashset. Therefore we have constant time lookup to see if a person is in a given room. For each edge we will look it up only once.

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 because you may have a negative cycle in which case if you include all the edges you do not have a tree. There is also the case where all the edges in a fully connected graph is negative in which case you definitely wouldn't have 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.

   Solution No. Kruskal's doesn't solve this problem because Kruskal's can't find negative cycles.

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 There are two ways we can do this. First we can select all the negative weight edges since we know that all the negative edges must be in the final lowest edge weight graph. You can always reduce the total weight by adding a negative edge. Once you find all of these edges you can then run Kruskal's algorithm to find a minimum spanning tree between these components to create a connected subset of edges.

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

Main Idea

There are a few primary concerns we have to take into consideration for this problem.

- Edges with fractional exchange rates have a similar function to negative edges since when you multiply a number by a fraction the result is smaller. Thus if we are to try to maximize the product, these fractional edges will create smaller weights.
- We also need to note that we aren't minimizing the path but instead maximizing the path.
- Lastly we are multiplying the edge weights instead of adding the edges.

We note that the presence of negative edge weights excludes the option of using Dijkstra's algorithm. Therefore we must use Bellman Ford to find the maximum path with the above modifications.

Psuedocode

```
def maxPath:
   for all c in currencies:
      weight(c) = inf
      prev(c) = nil
   dist(s) = 0
   repeat |c| - 1 times:
      for all c_i, c_j in rates:
         rate(c_j) = max{dist(c_j), dist(c_i) * rate(c_i, c_j)}

   return dist(t), createPath(t)
```

Run Time Analysis

The runtime of this algorithm will be proportional to the number of vertices in the graph. We need to update each edge with each iteration due to the fact that there are negative edges and thus we can't maintain the invariant from Dijkstra's that the path so far is the shortest to the given point. Also since we know that the graph is fully connected - there is an exchange rate between every two currencies - we determine that the runtime is $O(|V||E|)$.

Proof of Correctness

This algorithm is correct because with each update we are going in and modifying the weight if there is a newly found longer path to the given vertex. Thus we don't assume that we have found the longest path at any given point until we consider all possible paths to that vertex.

(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

Main Idea
We note that from the previous part we have 'negative edges' if an edge has a fractional weight. This is because the product with a fraction reduces the overall weight. When using Bellman Ford we know that if we run the iteration V times and see an update on the last iteration we have found a cycle. Therefore we run the algorithm V times and check the last update.

Psuedocode

```
def findArbitrage:
   for all c in currencies:
      weight(c) = inf
      prev(c) = nil
   dist(s) = 0
   repeat |c| - 1 times:
      for all c_i, c_j in rates:
         rate(c_j) = max{dist(c_j), dist(c_i) * rate(c_i, c_j)}
   for all c_i, c_j in rates:
      if dist(c_i) * rate(c_i, c_j) > max{dist(c_j):
         return True
   return False
```

Proof of Correctness
We have to prove that on the Vth iteration we will detect a cycle if one exists. Any path from s to t can have at most —V-1— edges because we assume there are no cycles. On the last update if there is still an update in the length then we know that there is a cycle because no path can have more than V edges.
Runtime Analysis
The runtime of this is the same as the previous part. We are running Bellman Ford on this and therefore will have a runtime of $O(V^2)$.