CS 170 HW 5 (Optional)

Due 2019-09-02, at 10:00 pm
You may submit your solutions if you wish them to be graded, but they will be worth no points

1 Study Group

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

2 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 \( i, j \).

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 \( r_{i_1,i_2} \cdot r_{i_2,i_3} \cdots r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1 \). 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 advantageous sequence of currency exchanges for converting currency \( s \) into 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).

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

\textbf{Proof of Correctness:}
To find the most advantageous ways to convert \( c_s \) into \( c_t \), you need to find the path \( c_{i_1}, c_{i_2}, \cdots, c_{i_k} \) maximizing the product \( r_{i_1, i_2}r_{i_2, i_3} \cdots r_{i_{k-1}, i_k} \). 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.

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

(b) \textbf{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.

\textbf{Pseudocode:}
This algorithm takes in the same graph constructed in the previous part.
1: function HASARBITRAGE(G)
2:   \( \text{dist, prev} \leftarrow \text{BellmanFord}(G, l, s) \)
3:   \( \text{dist}^* \leftarrow \text{Update all edges one more time} \)
4:   return True if for some \( v \), \( \text{dist}[v] > \text{dist}^*[v] \)

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

\textbf{Runtime:}
Same as Bellman-Ford, \( O(|V|^3) \).

\textbf{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.

3 Picking a Favorite MST

Consider an undirected, weighted graph for which multiple MSTs are possible (we know this means the edge weights cannot be unique). You have a favorite MST, \( F \). Are you guaranteed that \( F \) is a possible output of Kruskal’s algorithm on this graph? How about Prim’s? In other words, is it always possible to “force” the MST algorithms to output \( F \) without changing the weights of the given graph? Justify your answer. **Solution:** Yes; for both MST algorithms,
it’s possible to ensure they output $F$, provided it is indeed an MST.

First, consider Kruskal’s algorithm. Make sure that the edges of $F$ are always before any other equally-weighted edges after the sort. Now, it will add all such edges as early as possible. Consider towards a contradiction the case where at any point Kruskal’s declines to add an edge $e$ in $F$ to the MST. This means that $e$ would have created a cycle with other equally-weighted edges in $F$, and/or lighter edges (possibly in $F$). (Before this point, Kruskal’s may have added edges not in $F$ to the MST.) But then it’s impossible that both $e$ and all of the other equally-weighted edges in $F$ in this cycle can be in any MST, as one of them is the heaviest edge in some cycle of the graph, and such edges cannot be in any MST. Since we assumed that $F$ is an MST, this is a contradiction. Therefore we conclude that Kruskal’s algorithm will add all edges in $F$ to the MST. And since all MSTs have the same number of edges, this means it cannot add any edges not in $F$.

Now, consider Prim’s algorithm. As it expands the fringe, have it only choose edges in $F$ (so when there are multiple lightest edges to choose, choose a lightest edge in $F$). If this strategy fails, there must have been some cut across which none of the lightest edges were in $F$. But if this is the case, $F$ cannot have been an MST (one of the lightest edges across any cut must be in any MST). Given that $F$ must be an MST, this strategy will work.

### 4 Finding MSTs by Deleting Edges

Consider the following algorithm to find the minimum spanning tree of an undirected, weighted graph $G(V,E)$. For simplicity, you may assume that no two edges in $G$ have the same weight.

**procedure** `FindMST(G(V,E))`

1. $E' \leftarrow E$
2. for Each edge $e$ in $E$ in decreasing weight order do
3.     if $G(V,E'-e)$ is connected then
4.         $E' \leftarrow E'-e$
5. return $E'$

Show that this algorithm outputs a minimum spanning tree of $G$.

**Solution:**

There are several solutions. One is to note that, anytime the algorithm chooses not to delete an edge $e$, that edge must be a lightest edge across some cut. In particular, the cut is the two components of the disconnected graph $E'-e$ (using the value of $E'$ at the start of the iteration where the algorithm looks at $e$). The algorithm is also guaranteed to output $E'$ containing no cycles, so by applying the cut property we get that it outputs a minimum spanning tree.

Other proofs include the use of the cycle property. As a review, the cycle property claims that the heaviest edge in any cycle in $G$ cannot appear in the (unique) minimum spanning tree. To prove the cycle property, suppose $e$ is the heaviest edge in some cycle $C$ and is in the minimum spanning tree $T^*$. Consider deleting $e$ from the minimum spanning tree to get $T^*-e$. $T^*-e$ has two components, and some edge $e'$ in $C$ other than $e$ must connect the two components. So $T^*-e+e'$ is a spanning tree, and costs less than $T^*$ since $e'$ costs less than $e$, a contradiction.
The algorithm is guaranteed to output a spanning tree $T$. Suppose that the MST $T^*$ is not $T$, and let $e \in T^*-T$. Then $T \cup e$ contains a cycle; denote by $e'$ its heaviest edge, and note that $e \neq e'$ by the cycle property. When we considered $e'$, all edges in $T \cup e$ were still there, and so we should have deleted $e'$ since removing it would leave the graph connected.

An alternative proof is to show that every edge we delete is the heaviest edge in some cycle. This is because whenever we delete an edge $e$, it is part of some cycle in the remaining edges in $E'$ since $E'$ remains connected after deleting $e$. No other edge in this cycle can be heavier than $e$, otherwise we would have deleted that edge first. So by the cycle property we have only deleted edges that do not appear in the minimum spanning tree. Furthermore, note that this algorithm will eliminate all cycles from $T$. So we know the final solution is a tree, and thus must be the minimum spanning tree.

5 Unique Shortest Path

Shortest paths are not always unique: sometimes there are two or more different paths with the minimum possible length. Show how to solve the following problem in $O((|V| + |E|) \log |V|)$ time.

**Input:** An undirected graph $G = (V, E)$; edge lengths $l_e > 0$; starting vertex $s \in V$.

**Output:** A Boolean array $usp[\cdot]$; for each node $u$, the entry $usp[u]$ should be true if and only if there is a unique shortest path from $s$ to $u$. (Note: $usp[s] = \text{true}$.)

[Provide 3 part solution.]

**Solution: Main Idea:**
Suppose there are two different shortest paths from $s$ to $u$. These two paths can either share the same last edge (the edge ending at $u$), or not. If they do, this can be detected by modifying Dijkstra’s to detect if a node has been added to the known region previously with the same distance. If not there must be two different shortest paths to $u$’s parent, which can be detected by propagating (for every edge $(a, b)$) $usp(a)$ to $usp(b)$ if $\text{decreasekey}(H, v)$ is called.

**Pseudocode:**
This can be done by slightly modifying Dijkstra’s algorithm. The array $usp[\cdot]$ is initialized to true in the initialization loop. The main loop is modified as follows (lines 7-9 are added):

1. while $H$ is not empty do
2.   $u = \text{DELETEMIN}(H)$
3.   for all $(u, v) \in E$ do
4.     if $\text{dist}(v) > \text{dist}(u) + l(u, v)$ then
5.       $\text{dist}(v) = \text{dist}(u) + l(u, v)$
6.       $\text{decreasekey}(H, v)$
7.       $usp[v] = usp[u]$
8.     else if $\text{dist}(v) = \text{dist}(u) + l(u, v)$ then
9.       $usp[v] = \text{false}$

**Proof of Correctness:**
By Dijkstra’s proof of correctness, this algorithm will identify the shortest paths from the source $u$ to the other vertices. For uniqueness, we consider some vertex $v$. Let $p$ denote the shortest path determined by Dijkstra’s algorithm. If there are multiple shortest paths,
then take another path $p' \neq p$ and it will either share the same final edge $(w, v)$ (for some vertex $w$) as $p$, or they have different final edges from $p$. In the former case, there must be multiple shortest paths from $u$ to $w$. Using an inductive argument, which supposes that $\text{usp}$ is already set correctly for all vertices that are closer than $v$, this will be detected in the first conditional statement when the algorithm explores from $w$ and updates the distance to $v$ by taking edge $(w, v)$, as it will detect that there are multiple shortest paths to $w$. In the latter case, if the last edges of the two shortest paths are $(w_1, v)$ and $(w_2, v)$, then since edge lengths $l_e > 0$, both $w_1$ and $w_2$ must be visited before $v$, thus the algorithm will detect the existence of multiple shortest paths with the second conditional statement. The base case is true because there is a unique shortest path from the source $s$ to itself.

**Runtime:**
The runtime analysis follows that of Dijkstra’s, and will run in the required time when a binary heap is used for the priority queue.

### 6 Service scheduling

A server has $n$ customers waiting to be served. Customer $i$ requires $t_i$ minutes to be served. If, for example, the customers were served in the order $t_1, t_2, t_3, \ldots, t_n$, then the $i$-th customer would wait for $t_1 + t_2 + \cdots + t_i$ minutes.

We want to minimize the total waiting time

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i).$$

Given the list of the $t_i$’s, give an efficient algorithm for computing the optimal order in which to serve the customers.

**Solution:** We use a greedy strategy, by sorting the customers in increasing order of service times and serving them in this order. The running time is $O(n \log n)$.

To prove correctness, for any ordering of the customers, let $s(j)$ denote the $j$-th customer in the ordering. Then

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{s(j)} = \sum_{i=1}^{n} (n - i) t_{s(i)}.$$

For any ordering, if $t_{s(i)} > t_{s(j)}$ for $i < j$, then swapping the positions of the two customers gives a better ordering. Since we can generate all possible orderings by swaps, an ordering which has the property that $t_{s(1)} \leq \ldots \leq t_{s(n)}$ must be the global optimum. This is exactly the ordering that we output.