CS 170 HW 5

Due **2021-02-23, at 10:00 pm**

1 Study Group

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

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 $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$. 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 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:

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

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

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.

Pseudocode: This algorithm takes in the same graph constructed in the previous part.

function HasArbitrage(G)
  $\texttt{dist}, \texttt{prev} \leftarrow$ BellmanFord(G, l, s)
  $\texttt{dist}^* \leftarrow$ Update all edges one more time
  return True if for some v, $\texttt{dist}[v] > \texttt{dist}^*[v]$

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.

3 Money Changing.

Fix a set of positive integers called *denominations* $x_1, x_2, \ldots, x_n$ (think of them as the integers 1, 5, 10, and 25). The problem you want to solve for these denominations is the following: Given an integer A, express it as

$$A = \sum_{i=1}^{n} a_i x_i$$

for some nonnegative integers $a_1, \ldots, a_n \geq 0$.

(a) Under which conditions on the denominations x_i are you able to do this for all integers $A > 0$?

(b) Suppose that you want, given A, to find the nonnegative a_i's that satisfy $A = \sum_{i=1}^{n} a_i x_i$, and such that the sum of all a_i's is minimal —that is, you use the smallest possible number of coins. Define a *greedy algorithm* for this problem. (Your greedy algorithm may not necessarily solve the problem, i.e., it may give a suboptimal answer on some inputs)

(c) Show that the greedy algorithm finds the optimum a_i's in the case of the denominations 1, 5, 10, and 25, and for any amount A.

(d) Give an example of a denomination where the greedy algorithm fails to find the optimum a_i's for some A. (Do you know of an actual country where such a set of denominations exists?)

Solution:

(a) A can be expressed as a linear combination of the x_i if and only if $x_i = 1$ for some i. If one of your denominations x_i is 1, you will certainly be able to express every integer A as $\sum_{i=1}^{n} a_i x_i$ for some nonnegative integers $a_1, \cdots, a_n$. Conversely, in order to express $A = 1$ as a linear combination, you must have $x_i = 1$ for some i.

(b) Order your denominations such that $x_1 > x_2 > \cdots > x_n$. Then the *greedy algorithm* for this problem would be: Given A, let a_1 be the largest integer such that $a_1 x_1 \leq A$. If $A - a_1 x_1 > 0$, let a_2 be the largest integer such that $a_2 x_2 \leq A - a_1 x_1$. If you have nothing left over after doing this for $i = 1, \cdots, n$, then $A = \sum_{i=1}^{n} a_i x_i$.

(c) Since 1 divides 5 and 5 divides 10, it is clear that if we have a case in which the greedy algorithm would not find the optimal solution, it must involve 25, *i.e.* A must be greater than 25.

Note that $x_4 = 1$ cent, $x_3 = 5$ cent, and so on.

Assume the greedy algorithm does not find the optimal solution for A, $A > 25$.

Then $A = \sum_{i=1}^{4} a_i x_i = \sum_{i=1}^{4} b_i x_i$ and $\sum_{i=1}^{4} a_i > \sum_{i=1}^{4} b_i$, where the a_i were determined by the greedy algorithm and the b_i are optimal in that $\sum_{i=1}^{4} b_i$ is minimal.

W.l.o.g. $a_4 = b_4$ [since $a_4 \leq 4$ any change of the number of 1 cent coins must occur in 5 unit steps to give the same sum-this is obviously worse than changing b_3]. Also, since the other denominations are $5, 10, 25$, the number of 1 cent coins that the optimal algorithm takes must be A mod 5, which is the number of 1 cent coins our greedy algorithm takes too. In addition to that note that $a_3 \leq 1$.

By the above considerations we must have $a_1 > b_1$. Why? Because our greedy algorithm can certainly not pick *less* 25-cent coins than the optimal algorithm. The first thing our greedy algorithm does is pick as many 25-cent coins as possible! Also, a_1 is not *equal* to b_1, because if it were, then we know that our greedy algorithm correctly picks the optimal set of coins until $A = 24$ anyway (since 1 divides 5 and 5 divides 10.)

So, let $x := a_1 - b_1$. Note that x is a positive number.

For a_2, b_2 have three cases to consider: $a_2 = b_2$, $a_2 > b_2$ and $a_2 < b_2$.

Let's set $y := a_2 - b_2$.

Now, remember that $a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 = b_1x_1 + b_2x_2 + b_3x_3 + b_4x_4$. We can rewrite this as $b_3 = 5x + 2y + a_3$, using the actual values of x_i, the fact that $a_4 = b_4$, and our definitions of x and y.

Thus the number of coins changes by $\sum_{i=1}^4 b_i - \sum_{i=1}^4 a_i = 4x + y$. If we can show that this number is positive, this is a contradiction and we are done, since we expected $\sum_{i=1}^4 a_i > \sum_{i=1}^4 b_i$.

In cases 1 and 2, x and y are ≥ 0. Therefore $4x + y$ is clearly positive.

In case 3, y is negative. But, as we have to ensure that $b_3 = 5x + 2y + a_3$ is ≥ 0 and we know that a_3 is at most 1, we have $y \geq -\frac{5}{2}x - \frac{1}{2}$. Hence $4x + y \geq \frac{3}{2}x - \frac{1}{2}$ and it is again positive.

(d) A couple of real world examples:

- The United States of America 1875 – 1878 had 25 cent, 20 cent, 10 cent and 5 cent coins (and no 40 cent coins). To get 40 cents, the *greedy* algorithm gives 25 — 10 — 5, *i.e.* three coins, whereas the minimum is two coins (20 — 20)
- Prior to the change to the decimal system, Britain and many of her colonies had the following system:

			1 shilling	=	12 pence
	1 florin	=	2 shillings	=	24 pence
1 half–crown		=	2.5 shillings	=	30 pence

  So to get 36 pence, the *greedy* algorithm would take a half–crown and six pennies (*i.e.* seven coins), whereas one florin and one shilling (two coins) would be the minimal solution
- Cyprus in 1901 had 18 Piastres, 9 Piastres, 4.5 Piastres and 3 Piastres Silver coins and 1 Piastre, 0.5 Piastre and 0.25 Piastre Bronze coins.
  To get 6 Piastres, the *greedy* algorithm would take 4.5, 1 and 0.5 Piastre coins (three coins), whereas the minimum would be two 3 Piastre coins
- Persia under Muzaffar–ed–din Shah (1896 – 1907) had the following coins: 2 Tomans (= 400 Shahi), 1 Toman (= 200 Shahi), 0.5 Toman (= 100 Shahi), 4 Kran (= 80 Shahi), 0.25 Toman (= 50 Shahi), 2 Kran (= 40 Shahi), 1 Kran (= 20 Shahi), 0.5 Kran (= 10 Shahi), 0.25 Kran (= 5 Shahi), 3 Shahi, 2 Shahi and 1 Shahi.
  To get the sum of 160 Shahi, the *greedy* algorithm would take a 100 Shahi, a 50 Shahi and a 10 Shahi coin (three coins), whereas the minimum would be two 80 Shahi coins

4 Bounded Bellman-Ford (Optional)

You may submit your solution to this problem if you wish it to be graded, but it will be worth no points.

Modify the Bellman-Ford algorithm so that given a graph G, nodes s and t, and an integer k, it finds the weight of the lowest-weight path from s to t with the restriction that the path must have at most k edges.

Solution: The obvious instinct is to run the outer loop of Bellman-Ford for k steps instead of $|V|-1$ steps. However, what this does is to guarantee that all shortest paths using at most k edges would be found, but some shortest paths using more that k edges might also be found. For example, consider a path on 10 nodes starting at s and ending at t, and set $k=2$. If Bellman-Ford processes the vertices in the order of their increasing distance from s (we cannot guarantee beforehand that this will **not** happen) then just one iteration of the outer loop finds the shortest path from s to t, which contains 10 edges, as opposed to our limit of 2. We therefore need to limit Bellman-Ford so that results computed during a given iteration of the outer loop are not used to improve the distance estimates of other vertices during the **same** iteration.

We therefore modify the Bellman-Ford algorithm to keep track of the distances calculated in the previous iteration.

Algorithm 1 Modified Bellman-Ford

Require: Directed Graph $G=(V,E)$; edge lengths l_e on the edges, vertex $s \in V$, and an integer $k>0$.
Ensure: For all vertices $u \in V$, $dist[u]$, which is the length of path of lowest weight from s to u containing at most k edges.

for $v \in V$ **do**
  $\texttt{dist}[u] \leftarrow \infty$
  $\texttt{new-dist}[u] \leftarrow \infty$
$\texttt{dist}[s] \leftarrow 0$
$\texttt{new-dist}[s] \leftarrow 0$
for $i=1,\dots,k$ **do**
  for $v \in V$ **do**
    $\texttt{previous-dist}[v] \leftarrow \texttt{new-dist}[v]$
  for $e=(u,v) \in E$ **do**
    $\texttt{new-dist}[v] \leftarrow \min(\texttt{new-dist}[v], \texttt{previous-dist}[u] + l_e$

Assume that at the beginning of the ith iteration of the outer loop, $\texttt{new-dist}[v]$ contains the lowest possible weight of a path from s to v using at most $i-1$ edges, for all vertices v. Notice that this is true for $i=1$, due to our initialization step. We will now show that the statement also remains true at the beginning of the $(i+1)$th iteration of the loop. This will prove the correctness of the algorithm by induction. We first consider the case where there is no path from s to v of length at most i. In this case, for all vertices u such that $(u,v) \in E$, we must have $\texttt{new-dist}[u] = \infty$ at the beginning of the loop. Thus, $\texttt{new-dist}[v] = \infty$ at the end of the loop as well. Now, suppose that there exists a path (not necessarily simple) of length at most i from s to v, and consider such a path of smallest possible weight w. We want to show that $\texttt{new-dist}[v] = w$.

Let u be the vertex just before v on this path. By the induction hypothesis, at the end of the loop on line 7, $\texttt{previous-dist}[u]$ stores the weight of the lowest weight path of length at most $i-1$ from s to u, so that when the edge (u, v) is proceed in the loop on line 9, we get $\texttt{new-dist}[v] \leq w$.

Now, we observe that at the end of the loop on line 9, we have

$$\texttt{new-dist}[v] = \min\left(\texttt{previous-dist}[v], \min_{u:(u,v)\in E}\left(\texttt{previous-dist}[u] + l_{(u,v)}\right)\right).$$

Note that by the induction hypothesis, each term in the minimum expression represents the length of a (not necessarily simple) path from s to v of length at most i. Thus, in particular, none of these terms can be smaller than w, so that $\texttt{new-dist}[v] \geq w$. Combining with $\texttt{new-dist}[v] \leq w$ obtained above, we get $\texttt{new-dist}[v] = w$ as required.