Currency Arbitrage with the Bellman-Ford Algorithm in Python

Gary Cook

September
2019

Let the element $a_{i,j} > 0$ in table T represent the amount of currency j that can be purchased for 1 unit of currency i

$$T = \begin{array}{c|cccc} & 0 & 1 & \dots & n-1 \\ \hline 0 & 1 & a_{0,1} & \dots & a_{0,n-1} \\ 1 & a_{1,0} & 1 & \dots & a_{1,n-1} \\ \vdots & & & \vdots & \\ n-1 & a_{n-1,0} & a_{n-1,1} & \dots & 1 \end{array}$$

Approach The table T_2 is created by changing each element as follows:

$$a_{i,j} \text{ to } -log((\frac{1}{1+0.01c}a_{i,j}))\forall i,j \in \{0,1,\dots,n-1\},$$

where c can represent commission or account for small rounding errors. The table T_2 can be used to represent a complete weighted digraph $G = (V, E)$, where the set of vertices $V = \{0, 1, \dots, n-1\}$ represents the currencies and the edge from currency i to currency j in E is denoted $e_{i,j}$. The weight $w_{i,j} = a_{i,j}$ is assigned to the edge $e_{i,j}$. The negative logarithm is used because

$$-log(k_0k_1\dots k_m) = -log(k_0) - log(k_1) - \dots - log(k_m)$$

and since

$$log(x) \begin{cases} < 0 & \text{if } x < 1 \\ = 0 & \text{if } x = 1 \\ > 0 & \text{if } x > 1 \end{cases}$$

it follows that $k_0k_1\dots k_m > 1$ if and only if $-log(k_0) - log(k_1) - \dots - log(k_m) < 0$.

Therefore, the problem of determining if the currencies $0, 1, \dots, n-1$ can yield an arbitrage profit is equivalent to the problem of determining if the graph G contains a negative cycle, that is, a cycle where the sum of the weights assigned to its edges is zero. The Bellman-Ford algorithm is an algorithm that is used to compute the shortest paths from a source vertex to every vertex in a weighted digraph. Here shortest does not refer to the number of vertices in the path, but to the smallest sum of weights assigned to its edges. The Bellman-Ford algorithm is used because it has the ability to determine if G contains a negative cycle. However, in the paper Negative-Weight Cycle Algorithms[X. Huang], Algorithm A provides a method of updating the Bellman-Ford algorithm to find a negative cycle in G, if one exists.

Code

Bellman-Ford Algorithm The function $bellman\_ford\_algorithm$ implements the Bellman-Ford algorithm for G. The Bellman-Ford algorithm uses two vectors *distance* and *predecessor*, both of length n, where the i-th element of *distance* is the distance [=sum of weights] of the i-th vertex from the source along the currently shortest known path from the source vertex to the i-th vertex and the i-th element of *predecessor* is the vertex preceding the i-th vertex on that shortest path. The Bellman-Ford algorithm has three parts; as follows:

(a) With the exception of $distance[source]$, which is set to zero, all of the elements of *distance* are set to infinity and all of the elements of *predecessor* are set to null.

(b) For every pair $v_i, v_j \in V$, if $distance[j]$ is greater than $distance[i]+w(e_{j,i})$, then $distance[j]$ is updated to $distance[i] + w(e_{j,i})$ and $predecessor[j]$ is updated to i. This process is repeated $n-1$ times and at the k-th iteration [starting from 0] the algorithm finds all the shortest paths with at most $k+1$ edges [without cycles].

(c) Since a path without cycles can have at most $n-1$ edges, the process above is repeated and if it is possible to update an edge, then a path with n edges has been identified which is only possible if the graph contains at least one negative cycle.

Algorithm A If the Bellman-Ford algorithm determines that G contains a negative cycle, then the function $find\_cycle$ implements the update from Algorithm A [X. Huang] which is used to find a negative cycle. This is done by working backwards through the vector *predecessor* starting with the vertex that can be updated in 3(c) and continuing until a repetition is discovered.