22 SPRING CSCE 629 600: ANALYSIS OF ALGORITHMS - Homework 5

Name: Rohan Chaudhury

UIN: 432001358

Question (1) Textbook page 679, Problem 24-3.

Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy 49x2x0.0107 = 1.0486 U.S. dollars, thus turning a profit of 4.86 percent.

Suppose that we are given n currencies $c_1, c_2, \ldots . c_n$ and an n x n table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j .

a. Give an efficient algorithm to determine whether or not there exists a sequence of currencies $< c_{i1}, c_{i2}, \ldots . c_{ik} >$ such that
$R[i_1, i_2]. R[i_2, i_3]. R[i_3, i_4] \ldots . R[i_{k-1}, i_k] R[i_k, i_1] > 1.$
Analyze the running time of your algorithm.
b. Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

Answer (a):

Main idea of the algorithm:

In order to solve this problem, let us consider a graph G(V, E).

Let the vertices of the graph represent the given currencies $\boldsymbol{c_1, c_2, \ldots . c_n}$ and the directed edges between these vertices be represented as $(\boldsymbol{c_i, c_j})$. $\boldsymbol{R[i, j]}$ is the units of currency $\boldsymbol{c_j}$ that can be bought with one unit of currency $\boldsymbol{c_i}$**.** Then in the Graph G(V,E), we have V=$\{\boldsymbol{c_1, c_2, \ldots . c_n}\}$ as the vertices and E=$\{(\boldsymbol{c_i, c_j})\}$ as the edges for all the values of $\boldsymbol{R[i, j]}$ in the n x n table R.

Now, we have to determine whether or not there exists a sequence of currencies $< \boldsymbol{c_{i1}, c_{i2}, \ldots . c_{ik}} >$ such that

$$\boldsymbol{R[i_1, i_2]. R[i_2, i_3]. R[i_3, i_4] \ldots . R[i_{k-1}, i_k]. R[i_k, i_1] > 1}$$

We can use Bellman-Ford algorithm to solve this problem but in order to do that we have to determine the proper edge weights for the edges of this graph:

Given,

$$\boldsymbol{R[i_1, i_2].R[i_2, i_3].R[i_3, i_4] \ldots . R[i_{k-1}, i_k].R[i_k, i_1] > 1}$$

Let us take log on both sides:

$$=> log(R[i_1, i_2].R[i_2, i_3].R[i_3, i_4] \ldots . R[i_{k-1}, i_k].R[i_k, i_1]) > 0$$

$$=> log(R[i_1, i_2]) + \log(R[i_2, i_3]) + \cdots + \log(R[i_{k-1}, i_k]) + \log(R[i_k, i_1]) > 0$$

Multiplying by -1 on both sides we get:

$$=> -log(R[i_1, i_2]) - \log(R[i_2, i_3]) - \cdots - \log(R[i_{k-1}, i_k]) - \log(R[i_k, i_1]) < 0$$

So, if we assign the edge weights of the edges $(\boldsymbol{c_i, c_j})$ as $\boldsymbol{-\log(R[i_i, j])}$, then from the above equation we can see that the given problem has been converted to finding whether there is a negative cycle in the graph G(V, E) which has $\boldsymbol{-\log(R[i_i, j])}$ as edge weights for all edges $(\boldsymbol{c_i, c_j})$. Let us add an initial source vertex **src** in the graph G which is connected to all the other vertices of the graph and has edge weight=0. Now we can run **Bellman-Ford algorithm** from the source vertex $\boldsymbol{src}$ to find if there are any negative cycles in the graph G and return True if such a negative cycle exists.

<u>Pseudocode for the Bellman Ford Algorithm: BF(G, w, src)</u>

```
# src is the source vertex which is connected to all other vertices with edge weight 0

# G.V is the list of all vertices, G.E is the list of all edges of the graph G, w(x, y) gives the edge weight of edge (x,y)

# let us initialize an array dist of size equal to the number of vertices denoting distances from source src to all other vertices with the value infinity

for i in G.V:

        dist[i]=float("infinity")

# distance of source from source is 0

dist[src]=0

# Now we relax all edges |G.V| - 1 times.

for i in range(|G.V|-1):

        for each edge (x, y) in G.E:

                if dist[x] + w(x,y)< dist[y]:

                        dist[y]= dist[x] + w(x,y)


# Now we check if there are any negative cycles present in the graph

# after relaxing all edges for |G.V| - 1 times if for any edge (x,y) we get dist[x] + w(x,y)< dist[y] then there exists a negative cycle

#rest of the algorithm is in next page
```

```
for each edge (x, y) in G.E:
        if dist[x] + w(x,y)< dist[y]:
                #negative cycle exists, that is the required Arbitrage is possible
                return True
# otherwise return false
return False
```

Proof of correctness:

Bellman-Ford algorithm gives the shortest path from source to all the other vertices of a graph G(V, E) after|G.V|-1 relaxations. After relaxing all edges for |G.V| - 1 times if for any edge (x,y) we get distance[x] + weight(x,y)< distance[y] then our algorithm would return true indicating that a negative-cycle exists in the graph G. Then in such a case we get:

$$-log(R[i_1, i_2]) - \log(R[i_2, i_3]) - \cdots - \log(R[i_{k-1}, i_k]) - \log(R[i_k, i_1]) < 0$$

(since we assigned the edge weights of edges $(\boldsymbol{c_i}, \boldsymbol{c_j})$ as $-\mathbf{log}\ (\boldsymbol{R}[\boldsymbol{i_i}, \boldsymbol{j}])$),

the above equation can be rewritten as:

$$=> log(R[i_1, i_2].R[i_2, i_3].R[i_3, i_4] \ldots . R[i_{k-1}, i_k].R[i_k, i_1]) > 0$$

$$\boldsymbol{or} => \boldsymbol{R}[\boldsymbol{i_1}, \boldsymbol{i_2}].\boldsymbol{R}[\boldsymbol{i_2}, \boldsymbol{i_3}].\boldsymbol{R}[\boldsymbol{i_3}, \boldsymbol{i_4}] \ldots . \boldsymbol{R}[\boldsymbol{i_{k-1}}, \boldsymbol{i_k}].\boldsymbol{R}[\boldsymbol{i_k}, \boldsymbol{i_1}] > \mathbf{1}$$

which is our required solution indicating that the required Arbitrage is possible.

Time Complexity:

If we have n number of vertices (V) in the graph then the total number of edges (E) is given by n*(n-1)/2. We know that Bellman-Ford has a time complexity of O(V.E) since it performs the relaxation step |V|-1 times for all the edges E of the graph G. So in this case, the time complexity of the algorithm would be:

O(V.E)=O(n*n*(n-1)/2))=**O(n^3)**

Hence the total time complexity of the algorithm would be **O(n^3).**

Answer (b):

Main idea of the algorithm:

In order to print out a sequence if such a sequence exists in which Arbitrage is possible we need to modify the above algorithm to store the parent nodes of all the nodes while we are relaxing all edges |V| - 1 times. If a negative cycle is detected at a vertex y then we print the vertex y and then its parent node in the negative cycle and go on printing the corresponding parent nodes in the negative cycle till we again reach the node y in the cycle. The modified algorithm is shown below:

Pseudocode for the Bellman Ford Modified Algorithm: BFM(G, w, src)

```
# src is the source vertex which is connected to all other vertices with edge weight 0

# G.V is the list of all vertices, G.E is the list of all edges of the graph G, w(x, y) gives the edge weight of edge (x,y)

# let us initialize an array “dist” of size equal to the number of vertices denoting distances from source src to all other vertices with the value infinity

# let us initialize an array “parent” of size equal to the number of vertices which stores the parent of all the nodes. Initially we assign all the values of the “parent” array to null.

for i in G.V:
        dist[i]=float(“infinity”)
        parent[i]=NULL

# distance of source from source is 0

dist[src]=0

# Now we relax all edges |G.V| - 1 times and also store the parent values of the nodes while relaxing

for i in range(|G.V|-1):
        for each edge (x, y) in G.E:
                if dist[x] + w(x,y)< dist[y]:
                        dist[y]= dist[x] + w(x,y)
                        parent [y]=x


# Now we check if there are any negative cycles present in the graph

# after relaxing all edges for |G.V| - 1 times if for any edge (x,y) we get dist[x] + w(x,y)< dist[y] then there exists a negative cycle

for each edge (x, y) in G.E:
        if dist[x] + w(x,y)< dist[y]:
        #negative cycle exists, that is the required Arbitrage is possible, so we print the cycle by calling the PRINT_SEQUENCE function defined in the next section
                PRINT_SEQUENCE(y, parent)

# otherwise return false

return False
```

Pseudocode for the Printing the Arbitrage sequence: PRINT_SEQUENCE (y, parent)

```
#This function is called if an arbitrage is detected in the above modified Bellman-Ford algorithm

# y is the vertex where the algorithm detected a negative cycle, "parent" is the list containing the parent nodes of all the nodes in the Graph G

# We first print the y vertex and then proceed to print all the predecessors in the cycle till we reach y again


print (y)


# we find the parent of y using the parent list


par=parent[y]


# running while loop till current parent node is not equal to the starting node y


while par != y:
        #if current parent node not equal to starting node y we print the current parent node
        print (par)
        #then we update current parent node to parent[current parent node] and continue in the loop
        par=parent[par]
```

Proof of correctness:

The **BFM(G, w, src)** algorithm is storing the parent nodes of all the nodes while we are relaxing all edges |V| - 1 times. Now after relaxing all edges |V| - 1 times, if we attempt to relax all the edges again and receive another vertex (let's say y) where relaxation is possible then it means that we have obtained a negative cycle in the graph. Now if we can print this cycle then that gives our required sequence where Arbitrage is possible. When we are receiving such a node y which is part of a negative cycle we are calling another function PRINT_SEQUENCE(y, parent) which takes y and the parent list as input and prints out all the nodes in the cycle by traversing the cycle backward starting from the node y by using the parent list. The function first prints the node y and then prints its parent node and keeps on printing the corresponding parent nodes in the negative cycle till the starting node y in the negative cycle is reached. The resulting printed nodes gives our required sequence where Arbitrage is possible.

Time Complexity:

If we have n number of vertices (V) in the graph then the total number of edges (E) is given by n*(n-1)/2. The **BFM(G, w, src)** algorithm has a time complexity of O(V.E) since it performs the relaxation step |V|-1 times for all the edges E of the graph G. So in this case, the time complexity of the algorithm would be:

O(V.E)=O(n*n*(n-1)/2))=**$O(n^3)$**

The PRINT_SEQUENCE function has one loop which will run a maximum of |V| number of times. So, the time complexity of the PRINT_SEQUENCE function is O(n).

Hence the total time complexity of the entire algorithm would be **$O(n^3)$.**