CS535 HomeWork-1 Solutions

1) **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 49 2 0:0107 = 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 n × n table R of exchange rates, such that one unit of currency ci 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} ;. . .; c_{ik} such that $R[i_1,i_2] \cdot R[i_2,i_3] \cdot \cdot \cdot R[i_{k-1},i_k] \cdot R[i_k,i_1] > 1$. 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

Solution:

a) In the above problem description each currency $c_1, c_2 \ldots c_n$ can be considered as vertex V i.e |V| = n and the exchange rate between them is considered as edge E i.e $|E| = \binom{n}{2}$. We have to find the product of edge weights $R[i_1,i_2] \cdot R[i_2,i_3] \cdot \cdot \cdot R[i_{k-1},i_k] \cdot R[i_k,i_1] > 1$ which means checking whether there exists a cycle which is greater than one.

$$R[i_1,i_2] \cdot R[i_2,i_3] \cdot \cdot \cdot R[i_{k-1},i_k] \cdot R[i_k,i_1] > 1$$

$$\frac{1}{R[i1,i2]} \cdot \frac{1}{R[i2,i3]} \ldots\ldots\ldots \frac{1}{R[ik,i1]} < 1$$

Taking logs on both sides

$$\log\left(\frac{1}{R[i1,i2]} \cdot \frac{1}{R[i2,i3]} \ldots\ldots\ldots \frac{1}{R[ik,i1]}\right) < 0$$

$$\log\left(\frac{1}{\mathrm{R[i1,i2]}}\right) + \log\left(\frac{1}{\mathrm{R[i2,i3]}}\right) + .. + \log\left(\frac{1}{R[ik,i1]}\right) < 0$$

That is sum of edges in the cycle should not be negative. So, now the problem has been converted to detecting negative cycle. We can use the algorithms which are known to us.

Therefore weight W (V_i, V_j) = $\log\left(\frac{1}{\mathrm{R[i,j]}}\right)$ =-log(R[i , j]).

We can use bellman-ford algorithm to detect the negative cycle in the graph of currency exchange rate.

Algorithm:

Step – 1: Introduce a vertex t with edge weight 0 and connect it to all the vertices of V.

Step – 2: Relax the edges till N-1 iterations.

Relax(u,v)

d(v)>d(u)+w(u,v)

then d(v)=d(u)+w(u,v)

predecessor d = u (predecessor graph d_{i-1})

Step – 3: In the N^{th} iteration, if d_n (V)= d_{n-1}(V) then there is no negative cycle

Step – 4: else if d_n (V) < d_{n-1}(V) then there exists a cycle.

Here d_i (V) = min { d_{i-1}V, $\frac{min}{(u,v)\in E}$ (d_{i-1}(u) + w(uv) } considering there exists an edge between u and v in the graph.

Running time of Algorithm:

To create a Graph with |E| = $\binom{n}{2}$ it takes n^2 time and to detect the negative cycle algorithm runs for n vertices which is n^3 therefore the running time will be O(n^3) to create and detect the negative cycle of the above graph.

b) In order to print the sequence.

By considering the predecessor graph when there is a change in the value of any vertex then we are concluding that there exists a negative cycle. So whenever you are traversing the predecessor graph, while finding a cycle if you reach the vertex which has already been seen in this chain then you stop and output the vertices till that vertex.

The running time will be same $O(n^3)$ because to print the vertices it is O(n) at max which will be $O(n^3)+O(n)$ that is equivalent to $O(n^3)$.

2) **Suppose that we are given n intervals $[x_i;y_i]$ for $1\leq i \leq n$, which together cover an interval $[x_0;y_0]$. Each interval $[x_i;y_i]$ has a positive cost c_i. Give an efficient algorithm to a subset of intervals with minimum total cost which together still cover the interval $[x_0;y_0]$.**

Solution:

Step -1 :We have to find the set of interval that covers $[x_0 , y_0]$ that is the intiial interval in the path.

Step-2:To do this we can use length/cost ratio. While trying to do this recursively to create the $[x_p;y_0]$ from the interval $[x_p;y_p]$ we might encounter set of intervals or only one interval.

Step -3: If there is only one interval before the $[x_p;y_p]$ then no need to compute any length/cost ratio .

Step-4 : else if, calculate the length/cost ratio to reach each of the interval from $[x_p;y_p]$ and chose the maximum ratio interval from the $[x_p;y_p]$. In this way we can reach the initial interval which we have encountered while traversing from $[x_p;y_p]$.

3) **Consider a digraph D = (V; A) with two distinct vertices s and t.**

(a) Let F denote the set of edges in A which appear in some shortest s-t path (in terms of the number of edges) in D. Give an algorithm to output F in O (|V| + |A|) time.

(b) Give an efficient algorithm to find an inclusion-wise maximal (not necessarily) edge-disjoint shortest s-t paths in D in O (|V | + |A|) time

Solution:

a) To find the shortest s-t path we use Breadth –First search algorithm. To get an algorithm of O (|V| + |A|) time we can assume the graph is unweighted graph because the BFS has the efficiency of O (|V| + |A|) time. Here we add extra memory to store the predecessor of a given vertex .

Algorithm:

Step 1: Initialize the array [0,1,…,v-1] with the distances of the each vertex from the source.

Step-2: Initialize the predecessor array for the every vertex v. we use this so that we need not visit the vertex which is already visited which makes we add the vertex till now which is visited and we add the vertex which comes in next and try for the coming next vertex to reach the destination.

Step-3: Run the normal BFS algorithm considering the above predecessor array.

For every new vertex coming in it checks whether it is visited or not. If it is not visited then we add it to the distance to already calculated path.

In this way we can get the distance of the vertex in O(1), printing the path will take O(V) and to run the BFS it takes O (|V| + |A|) time.

b)

Step -1 : from the above algorithm we will find the shortest path using the shortest path which was found using above algorithm as a first step.

Step-2 : Replace every edge of the shortest path with directed arc to the source vertex.

Step-3 : now every arc length should be changed to negative

Step-4: Run the shortest path algorithm which is above (as the above algorithm can take the negative edges too)

Step-5: now remove the overlapping edges in the both paths which we found in step-1 and step-4. Then replace the direction in such way that the direction to source vertex is now direction to target vertex.

Step-6: This will the shortest path with the edge-disjoint vertex.

As we are using the BFS algorithm which has O (|V| + |A|) time. This algorithm will also have the same time.

4) **Suppose that a digraph D = (V; A) with edge length function *l* has no negative circuit but has a 0-length circuit C. Let p be an arbitrary potential function, and l^* be the edge length function obtained by reweighting *l* with p. Prove that for any edge a $a \in C$, l^* (a) = 0**

Solution:

We know that D= (V,A) and l has no negative circuit but having 0-lenth circuit.

Consider a path P then the length of P l(P) is sum of all edge lengths in the path and distance d(u,v) .

P is an arbitrary potential function, we know that there does not exist a negative circuit so the potential function after reweighting the l with p it becomes

$l^*(P)=l(P) – p(u) +p(v)$. Now let us consider adding a new vertex to the D namely z and and connect this vertex to all the vertices in the graph.

Now $p(v) = d(z , v) \leq d(z , u) + l(u , v)$

Or $\leq p(u) + l(u,v)$ (By definition of potential function)

As per the property of potential function after the reweighting $l^*(u , v) \geq 0$

Hence, we can say that for any edge $a \in C$, $l^*(a) = 0$