CSCE 629-601 Homework 6
19th October, 2016

Name: Adil Hamid Malla **UIN:** 425008306

1. Exercise 22.3-3 (a) on page 640

INPUT: We have a Undirected $Graph = G(V, E)$. we define a Bottleneck Spanning tree as the spanning tree of G whose largest edge weight is minimum over all spanning trees of G. And the maximum weight edge is the value of the bottle neck spanning tree.
The problem is asking to show that the minimum spanning tree is the bottleneck spanning tree.

Solution:
Here since we have to show that the minimum spanning tree is same as the bottleneck spanning tree, so we have to prove this into two ways, one that the bottleneck spanning tree is minimum spanning and minimum spanning tree is bottleneck spanning tree. It is like if and only if condition proofs.
Lets say that we have a minimum spanning tree of our graph G. Lets denote it by MST. Now lets take the highest edge weight of this MST into considerations. Say the edge between u and v is of maximum weight i.e. $w(u, v) = Maxweight$. Now if we can show that this edge is also the highest edge in the bottleneck spanning tree then we are done. Now we lets move forward with the contradiction method:

1. Case 1: Lets assume the maximum weight in the bottleneck spanning tree is $w'(u', v') = Maxweight'$ and is greater than the $w(u, v)$ i.e. $Maxweight' > Maxweight$. But if this the case, we are contradicting with the definition of the bottleneck spanning tree by including the $w'(u', v') = Maxweight'$ edge, which can be replaced by the $w(u, v)$ to make the tree follow the definition of the bottleneck spanning tree.
2. Case 2: Lets assume the maximum weight in the bottleneck spanning tree is $w'(u', v') = Maxweight'$ and is less than the $w(u, v)$ i.e. $Maxweight' < Maxweight$. Then we have the case where we are not taking the edge which could further minimize the spanning tree as the edge $w'(u', v')$ which contradicts with our assumption the we have an edge whose weight is less than that of edge in MST tree. Thus it shows that we have $w'(u', v') = Maxweight'$ and is greater than the $w(u, v)$ i.e.
$Maxweight' < Maxweight$.

   Therefore we have showed that the MST is a bottle neck spanning tree with weight
   $w(u, v) = Maxweight$.

2. Problem 24-3 on page 679

INPUT: We have n currencies and the corresponding n by n table of R of exchange rates, such that one unit of the 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 $c_{i1}, c_{i2}, \ldots, c - ik$ i such that $R[i_1, i_2].R[i_3, i_4]\ldots.R[i_{k-1}, i_k].R[i_k, i_1] > 1$.

Solution: We construct a weighted directed graph $G = (V, E)$. Here each vertex represents each currency and edges (u, v) and (v, u) represents each pair of the currencies between two currencies.
Now we transform the problem statement into something which will be easy for us. If we go with the general brute force algorithm, we will check each cycle and calculate the product of the edges which should gives us the result more than 1, but this algorithm will take a lot of time. Can we do something to the problem statement to make it really easy for us to get the arbitrage which results in the profit. Now lets make the problem statement in such a way that we can use the graph algorithm on it which reduces the time complexity of our problem. Our original statement is :

$$R[i_1, i_2].R[i_3, i_4]\ldots.R[i_{k-1}, i_k].R[i_k, i_1] > 1.$$
$$=> 1/R[i_1, i_2].1/R[i_3, i_4]\ldots.1/R[i_{k-1}, i_k].1/R[i_k, i_1] < 1.$$
Taking log on both the sides
$$=> log(1/R[i_1, i_2]).log(1/R[i_3, i_4])\ldots.log(1/R[i_{k-1}, i_k]).log(1/R[i_k, i_1]) < 0$$

Thus we have transformed the problem into such a way that we redefined the weights of the edges so that now we are only looking for the negative cycles in the graph. For getting that we can use the Bellman-Fords Algorithm. So to apply the Bellman-Ford algorithm we will construct the directed graph from the R table. Thus, we can write the following algorithm to determine whether or not there exists a sequence of currencies presenting an arbitrage opportunity:

Algorithm 1 ARBITRAGE-CHECK

```
procedure ARBITRAGE-CHECK($R$)
    G = Make-Graph(R)
    if BELLMAN-FORD(G) = FALSE then
        return True
    end if
    return False
end procedure
```

Since the correctness is already ensured from the equations above. Now we will calculate the time complexity of our algorithm. Here making the graph G from the R table takes all the entries into consideration thus it takes $O\left(n^2\right)$. Afterwards, we also know that there are V vertices and the edges are V^2, as there are v edges for each vertex. Now we have applied the Bellman-Ford algorithm on our graph and we know that it takes $O\left(V * E\right)$ time. Thus the overall time complexity of checking if the arbitrage exists is $O\left(V * E\right)$ also equal to $O\left(V * V^2\right)$ that is $O\left(V^3\right)$

b. Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.
Solution: Now we can use the above algorithm to calculate the sequence of the currencies to get the arbitrage. Using the Bellman-Ford algorithm we can return false if some negative cycle appears in our graph. Thus it means we have to traverse that negative cycle to get the currencies resulting in the arbitrage.
Now lets check for what case our algorithm will result in false, it is when we have some edge between (u, v), then if we have $v.d > u.d + w\left(u, v\right)$, indicating the negative cycle that includes the u currency. Thus we follow the predecessor pointers from u until we encounter the u again making the finishing point of the cycle. The following algorithm will take care of printing the currencies which resulted in the arbitrage.
The algorithm's pseudo code is given:

Algorithm 2 PRINT-ARBITRAGE-CURRENCY

```
procedure ARBITRAGE-CHECK($R$)
    G = Make-Graph(R)
    if BELLMAN-FORD(G) = TRUE then
        return No Arbitrage Found
    end if
    for for each edge $(u, v) \in G.E$ do
        if $v.d > u.d + w\left(u, v\right)$ then
            $PRINT - PATH\left(G, u, u.\pi\right)$
            Return
        end if
    end for
end procedure
procedure PRINT-PATH($G, s, v$)
    if $v == s$ then
        print s
    else if $v.\pi == NIL$ then
        No Path
    else
        $PRINT - PATH\left(G, s, v.\pi\right)$
    end if
    print v
end procedure
```

Thus the overall time complexity of checking if the arbitrage exists is $O\left(V * E\right)$ also equal to $O\left(V * V^2\right)$ that is $O\left(V^3\right)$. The time taken to print the path is just $O\left(V^2\right)$.

3. Problem 26.1-7 on page 714

INPUT: We have a weighted directed Graph $G = (V, E)$ representing the flow network, additional to the edge capacity we also have the node/vertex capacity. We represent the network capacity of the vertex v by $l\left(v\right)$.
OUTPUT: Show how to transform a flow network G = V,E with vertex capacities into an equivalent flow network G'= V',E' without vertex capacities, such that a maximum flow in G' has the same value as a

maximum flow in G. How many vertices and edges does G' have?

Solution: The answer to this question is very tricky. We have to represent the flow network in such a way that the flow remains same in both the graphs. We are also given the constraint that the G' has to be without the vertex capacities, so we have to find the way of making the vertex capacities into the flow diagram. The obvious choice we can get is that we can change the vertex capacities into the edge capacity, now the algorithm to do that is defined below. For each vertex $v \in V$ we will replace the vertex v by vertex v and v' and the edge between them with the edge capacity equal to the vertex capacity of vertex v. Moreover all the incoming edges to vertex v are kept the same and all the outgoing edges are taken out from the vertex v'. Thus in short we are augmenting the vertex capacity by having an additional edge between original vertex v and new vertex v'. The Algorithm is defined as:

Algorithm 3 Vertex-Capacity Graph Transformation

```
1: procedure GRAPH-TRANSFORMATION($G$)
2:     for all the $v \in V$
3:     insert a vertex v' with an edge from $v \to v'$ of capacity $l(v)$
4:     all the outgoing edges from vertex v now goes out from vertex v'
5:     return the new graph G'
6: end procedure
```

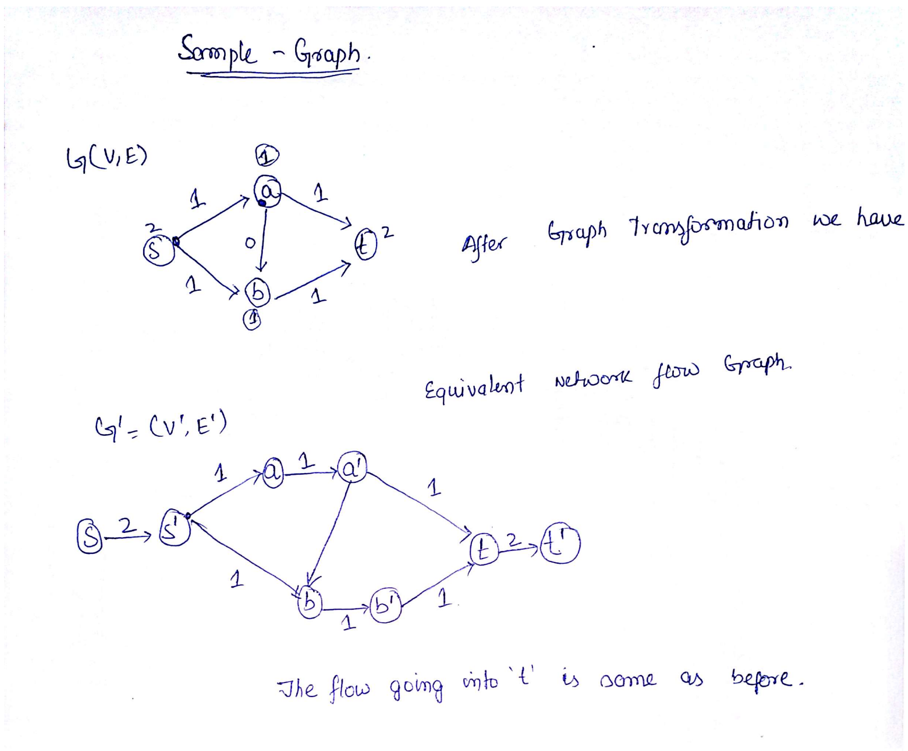
Sample - Graph.
G(V,E)
1
1
2
S
a
0
1
1
t
2
b
1
After Graph Transformation we have
Equivalent network flow Graph.
G' = (V', E')
S
2
S'
1
a
1
a'
1
1
b
1
b'
1
t
2
t'
The flow going into 't' is same as before.

Figure 1: Figure to Explain The Algorithm

Looking at the Figure 1. we came to conclusion that the network flow in the both cases are same. We are just replacing the vertex capacity by the equivalent edge with the same capacity.
Since by the transformation of the graph, we have changed the number of the vertices and number of edges. Now following the algorithm we can see that we are adding the V vertices in the graph G' and also we are adding equivalent number of edges in the graph which is V edges.
Thus in our final graph we have following numbers:
Number of edges in $G' = |E| + |V|$ Number of the vertex in $G' = |V| + |V| = 2 * |V|$
The running time of this algorithm will be $O(V)$ as we using the number of vertices to be resolved.