Assignment 4

1. Write the pseudocode of this algorithm and explain its core idea and critical procedures in a clear way.

```
Improved-BFM(G,s,t){
    foreach node v in V{
        M[v] = inf;
        first[v] = null;
        v.dormant = false;
    }
    M[t] = 0;
    Queue q;
    q.enqueue(t);
    while(!q.isEmpty()){
        v = q.dequeue();
        foreach node u in V s.t. (u,v) in E{
        if(!v.dormant and (v.subtree.contains(u) and u.dormant or
!v.subtree.contains(u))){
            if(M[u] > M[v] + c(u,v)){
                M[u] = M[v] + c(u,v);
                first[u] = v;
                u.dormant = false;
                if(u not in v.subtree)v.subtree.add(u);
                if(negLoop(u,v)){
                    return false;
                }
                sleep(u);
                if(!q.contains(u)){
                    q.enqueue(u);
                }
            }
        }
        }
    }
    return true;
}

negLoop(u,v){
    if (u.subtree.contains(v)){
        return true;
    }else{
        return false;
    }
}

sleep(u){
    foreach node v in u.subtree{
        if(first[v] == u){
            v.dormant = true;
            first[v] = null;
            sleep(v);
```

```
        }else{
            u.subtree.delete(v);
        }
    }
}

##pay attention to the function v.subtree.contains(u) like this
v.subtree.contains(u){
    cur = first[u]
    while(cur != null){
        if(cur == v){
            return true
        }
    }
    return false;
}

```

Core idea: Realize a subtree structure of each node to record all the nodes whose shortest path passes it. Actually, every node just need to maintain a list of nodes whose value of first is it (in pseudo-code we name it "subtree"). For example, if first[a]=first[b]=first[c]=d in final result then d keeps a list called subtree which contains a,b,c. Why not keep a real tree of all the nodes whost shortest path passing the root node? Because we only need to mark all these nodes as dormant in sleep() procedure, and we can use the list we keep in each node to implement it through recurssion. We can also use the list to find whether the node in one's subtree or not through recurssion. Thus, a real structure of tree is no need to keep in each node which instead will cost huge space complexity.

What's the benefit of keeping this tree? First, we can judge whether the u.subtree.contains(v) when (u,v) is is taken into consideration which means current u's shortest path passes v. If so, then a negtive loop is detected the algorithm is done. If not, we can prune the edges when a node is dormant which speeds up the the dp procedure.

Critical procedures: We will look at each function in detail.

negLoop(u,v): it is used to terminate the main function when detecting a negative loop in some node's shortest path to the node t.

sleep(u): it is used to set all of the nodes whose current shortest path passing node u as dormant. That means as M[u] is changed, their shortest may not still pass node u, which needs to be dynamic programed again. Meanwhile, we do the deletion of those nodes in subtree which actually have found another shorter path passing other nodes. Our aim is to maintain u.subtree as a list of all the nodes whose current shortest paths passing u, for each u in V.

v.subtree.contains(u): it is used to decide whether node u's current shortest path passing node v.

Improved-BFM(G,s,t): it's the main function of the algorithm. First, we initialize M[] and first[] and set all of the nodes as not dormant. Then, we begin from the node t and use queue structure to maintain the order of traversing nodes. It seems like BFS. We traverse all of the node which has a entering edge into current dequeued node v. And if v is dormant, that means M[v] is currently not the shortest distance from v to t, thus we can not perform dp on this node. If v is not dormant, there are two situaions when a node u with entering edge into v should be taken into consideration. On one hand, u is not in v.subtree, which means u haven't tried the path which passes v and probably after M[v]'s change u will find a shorter path passing v. On the other hand, u is in v.subtree and u is dormant, which means u's current shortest path passing v but M[u] is not the shortest path, so we can get the new shortest path of u which is currently still considered passing v. Then in above situation, we will update the new shortest path together with subtree of u then set it as not dormant and detect negative loop. Last, set all of the nodes whose current shortest path passing u as dormant and put u into queue if u has not been included in queue before. If a negative loop is detected then the main function will return false else true and then we can use first[] to get each node's shortest path.

---

2. Analyze its running time and space complexity.

Time complexity: Suppose |V|= n and |E| = m. The initialization costs O(n) obviously, and the rest codes in Improved-BFM(G,s,t) totally costs O(mn). Because each while we consider all of the nodes which have entering edge into current dequeue node and it costs O(m) at most.(b.t.w negloop(u) and sleep(u) both cost O(n) time at most, which is obvious. And m is more larger than n in general, so we consider it's O(m)).The procedure of algorithm is like BFS. Because we start from node t and add all of the nodes which have entering edge into t, which means we try to find shortest path of length 1 for these nodes. Then, we add all of the nodes which have enter edge into these nodes in certain order. It means we next try to find shortest path of length 2 for these new enqueue nodes. Therefore, during the whole procedure, we will find shortest path of at most length n-1 for each node. Thus, the total time complexity of rest part of Improved-BFM(G,s,t) is O(mn). Then, the total time complexity of whole algorithm is O(mn).

Space complexity: All the data structures we used like M[], first[], queue, and subtree is all one-dimensional array of at most size n. Thus, the space complexity of whole algorithm is O(n).

---

3. Explain in your words why in practice Tarjan's trick gains a considerable speedup compared to SPFA, even for graphs with no negative cycles. Note that the worst-case running time of this improved algorithm is nonetheless unchanged, i.e., O(mn).

The worst-running time of this improved algorithm is still O(mn). But it happens in the sparse gragh which means m is not bigger enough than n, like a list of nodes. However, in practice, m is much larger than n. And the improved algorithm implements a method like pruning to avoid take every edge into consideration in each iteration. We only care about the nodes which are not dormant. It speeds up SPFA to a great extent in dense gragh which means m>>n and the relationship between nodes is quite complex. Because SPFA will consider every edge entering the dequeue node in each iteration while improved-BFM is not. The principle of the speedup is not related to the value of edges which means this speedup can also play an enormous role in Shortest Path Problem of graphs without negative cycles.