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

The improvement can be separated into two part: **negative cycle detection** and **dormancy optimization**. Both of them are based on pointer graph, which record the information that the distance of each vertex is updated from whom.

Obviously, when there are no negative cycles, the whole pointer graph is a directed tree with this origin as the root, and each vertex is a root for its subtree.

As for negative cycle detection, while updating, taking $dist[v] = dist[u] + w$ as an example, if u is in the subtree of v, there forms a cycle, definitely negative. However, if we only use this property to check the negative cycle, it will cost a huge time to trace back the whole path for each update. But if we check cycle in the next iteration, there must be a edge that has been in the pointer graph being used again (an edge in the negative cycle). Now we could simply check whether an edge is chose once. So we can end the iteration procedure as soon as a negative cycle is found, rather than traditionally continuing till the iteration time of a node hit the upper bound of the length of a simple path.

Then comes the dormancy optimization. To reduce redundant useless iterations, here we use a sign called dormant. Also taking $dist[v] = dist[u] + w$ as an instance, clearly after v is updated by u, vertices in the subtree of v will all be updated from v sooner or later. However, some of they probably are in the queue now, waiting to be taken out and update their neighbors with outdated values. So we mark them as dormant, which means when encountering them, we can simply throw them back into the queue, until they are wakened up - updated by the new values.

```
void mark(int u)
{
    dormant[u] = true;
    for (v in son[u])
        if (fa[v] == u && !dormant[v])
            mark(v);
    son[u].clear();
}

void trace_back(int u, int s)
{
    if (u == s)
        return;
    negative_cycle.push_back(u);
    trace_back(fa[u]);
}

bool SP(int s)
{
    queue q;
    dist[s] = 0;
    q.push(s)
    while (!q.empty())
    {
```

```
        u = q.front();
        q.pop();
        if (dormant[u])
        {
            q.push_back(u);
            continue;
        }
        for (e in edge[u])
        {
            v = e.v;
            w = e.w;
            if (dist[u]+w < dist[v])
            {
                if (fa[v] == u)
                {
                    trace_back(v, v);
                    return false;
                }
                mark(v);
                dist[v] = dist[u]+w;
                dormant[v] = false;
                son[u].push_back(v);
                fa[v] = u;
                q.push(v);
            }
        }
    }
    return true;
}
```

2. Analyze its running time and space complexity.

From the analysis and pseudocode above, we can tell that both time and space complexity remain the same.

Analyzed in Q1, checking whether an edge is used only need $\Theta(1)$ extra time. Noticed that the dormancy state of a vertex can be inverted either its father is updated and itself has not be dormancy or itself is waken up when be updated, the time cost of marking is the same as that of updating distance, the extra time cost will be $O(m)$. So the upper bound of this optimized algorithm remains the same. But in the worst case, each time the algorithm choose the smallest larger edge to update the rest vertices and no dormancy flag is used, then it just performs like usual SPFA, so the upper bound is still $O(mn)$. (Optimization details are in Q3)

As for the space complexity, we need to maintain a pointer graph, which can be achieved by recording the father of each vertex. As every vertex can only has one father in the pointer graph, the space cost is $\Theta(n)$. And for the dormant flags, each vertex only needs one, so the space cost is also $\Theta(n)$. Thus, the general extra space cost is $\Theta(n)$, and $\Theta(n+m)$ for the whole algorithm.

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.

   One major time waste in the original SPFA algorithm is the case that, still using $dist[v] = dist[u] + w$, clearly after v is updated by u, vertices in the subtree of v will all be updated from v sooner or later. However, some of they probably are in the queue now, waiting to be taken out and update their neighbors with outdated values. And then their neighbors may pass these meaning values farther. Multiple time enqueue operations are also likely to exist. However, as elaborated in Q1, dormancy effectively avoid those snowball effects. Noticed that this trick has nothing to do with the exact value of edges, so there is no wonder that it also works for graphs with no negative cycles. But this trick only plays a role like pruning in searching, without guaranteeing the certain extent that can be optimized (or so called depending on the data instances). In the worst case (showed in Q2), it performs just like the plain SPFA, with time complexity $O(mn)$.