CS208 Theory Assignment 3

12312110 李轩然

DDL: Mar.18 20:50

Chapter 3 Exercise 1

Description

Consider the directed acyclic graph G in Figure 3.10. How many topological orderings does it have?

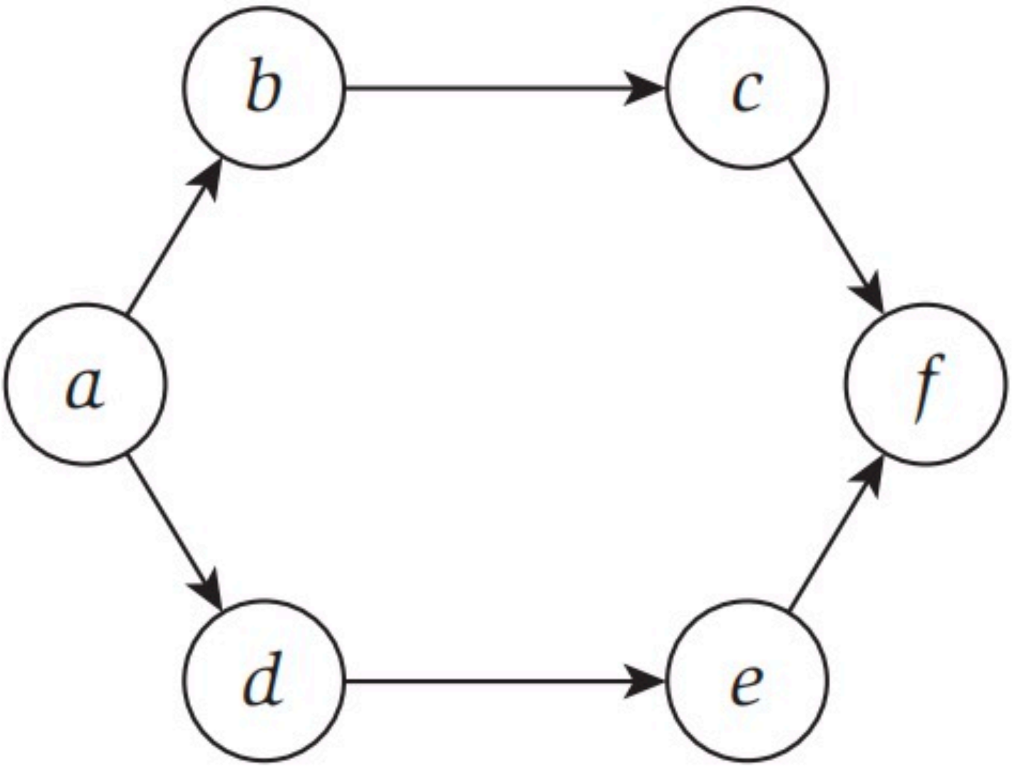

Figure 3.10 How many topological orderings does this graph have?

Analysis

Firstly, we find that G is a DAG, thus it has a topological ordering.

Then among all the nodes, only a has no incoming edges, and only f has no outcoming edges, thus a must be the first and f mast be the last.

For b can only be reached directly from a, and c can only be reached directly from b, and from c, we can only reach f, thus a must come before b, b must come before c, c must come before f.

Similiarly, a must come before d, d must come before e, e must come before f.

According to these conditions, we can list all the possible topological orderings:

$$a, b, c, d, e, f$$

$$a, b, d, c, e, f$$

$$a, b, d, e, c, f$$

$$a, d, b, c, e, f$$

$$a, d, b, e, c, f$$

$$a, d, e, b, c, f$$

Finally, we can conclude that we have six possible topological orderings.

Chapter 3 Exercise 3

Description

The algorithm described in Section 3.6 for computing a topological ordering of a DAG repeatedly finds a node with no incoming edges and deletes it. This will eventually produce a topological ordering, provided that the input graph really is a DAG.

But suppose that we're given an arbitrary graph that may or may not be a DAG. Extend the topological ordering algorithm so that, given an input directed graph G, it outputs one of two things: (a) a topological ordering, thus establishing that G is a DAG; or (b) a cycle in G, thus establishing that G is not a DAG. The running time of your algorithm should be $O(m+n)$ for a directed graph with n nodes and m edges.

Analysis

First, we construct the graph by 2-dimensional vector g. By g, we can count the number of incoming edges of every nodes, denoted as an array in. Now we find the node u which has no incoming edges ($in[u] == 0$) as a start. If we can not find this kind of node u, then it must have a cycle, and we do not need to do DFS but to search the cycle.

As we find all these us, we put them into the queue q, then do DFS:

1. Pop the front element u of q, add u to our topological ordering $topo$, and delete all the corresponding edges of u (that is, if there is a edge from u to i, then $in[i]$ should minus 1);
2. Find the new node v which has no incoming edges now and can be reached directly by u ($in[v] == 0$ and $v \in g[u]$), add all these vs to q and repeat these two steps;
3. The termination condition is, the queue q is empty.

After doing DFS, if G is a DAG, the size of $topo$ must equal to the number of nodes, and $topo$ is one valid topological ordering. Otherwise, G has a cycle and now we search it.

Notion that after topological ordering searching, every node in a cycle must have one or more incoming edges. That is because, if a node is in a cycle and has no incoming edges, it must be iterated, and then all the nodes in this cycle must be iterated, which makes a contridiction. Thus, we just need to find a node u that $in[u] > 0$, then traverse it until we meet u. The traversing path is a cycle.

Here is the method, and for every nodes and edges are iterated once, the time complexity will be $O(m + n)$.

```
vector<int> topological(const vector<vector<int>>& g, int n, int m) {
    vector<int> in(n, 0);
    vector<int> topo;
    queue<int> q;

    for (int i = 0; i < n; i++)
    {
        for (int v : g[i])
        in[v] ++;
    }

    for (int u = 0; u < n; u++)
```

```
    {
        if (in[u] == 0) q.push(u);
    }

    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        topo.push_back(u);
        for (int v : g[u])
        {
            in[v] --;
            if (in[v] == 0) q.push(v);
        }
    }

    if (topo.size() == n) return topo; else {
        vector<int> cycle;
        vector<bool> visited(n, true);

        int u, start;
        for (int i = 0; i < n; i++)
        {
            if (in[i] > 0) {
                start = i;
                break;
            }
        }

        u = start;
        while (visited[u])
        {
            cycle.push_back(u);
            visited[u] = false;
            for (int v : g[u])
            {
                if (in[v] > 0) {
                    u = v;
                    break;
                }
            }
        }

        cycle.push_back(start);
        return cycle;
    }
}
```