Analysis of Algorithms – HW2

Li-Hen Chen

Department of Computer Science & Engineering, Texas A&M University

1. Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge [u, v], vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

The question demands us to develop and $O(|V| + |E|)$ time algorithm using finding a vertex of in-degree 0, output it and removing it and all of its outgoing edges from the graph. The approach is based on the below fact:

A DAG G has at least one vertex with in-degree 0 and one vertex with out-degree 0.

- **Algorithm:**

Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.
Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)
Step-3: Remove a vertex from the queue (Dequeue operation) and then do
1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighboring nodes.
3. If in-degree of a neighboring nodes is reduced to zero, then add it to the queue.
Step 4: Repeat Step 3 until the queue is empty.
Step 5: If count of visited nodes is not equal to the number of nodes in the graph then the topological sort is not possible for the given graph (Has a cycle).

Pseudo-code:

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
```

```
            insert m into S
if graph has edges then
    return error     (graph has at least one cycle)
else
    return L     (a topologically sorted order)
```

Time Complexity:
The outer while loop will be executed V number of times and the inner for loop will be executed E number of times, Thus, overall time complexity is O(V+E). As a result, the overall time complexity of the algorithm is $O(|V| + |E|)$

2. A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. We can find all strongly connected components in O(V+E) time using following procedures.

1) Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack.
2) Reverse directions of all arcs to obtain the transpose graph.
3) One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS. The DFS starting from v prints strongly connected component of v. The SCC will pop out in same DFS if they are strongly connected. We can connect them by the pop out order to get the G^c, which represents the strongly connected component graph.

Pseudo-code:

```
1. For each vertex u of the graph, mark u as unvisited. Let L be empty.
2. For each vertex u of the graph do Visit(u), where Visit(u) is the recursive subroutine:
    If u is unvisited then:
        1. Mark u as visited.
        2. For each out-neighbor v of u, do Visit(v).
        3. Prepend u to L.
    Otherwise do nothing.
3. For each element u of L in order, do Assign(u,u) where Assign(u,root) is the recursive subroutine:
```

```
If u has not been assigned to a component then:
    1. Assign u as belonging to the component whose root is root.
    2. For each in-neighbor v of u, do Assign(v,root).
Otherwise do nothing.
```

Time Complexity:
Provided the graph is described using an adjacency list, the algorithm performs two complete traversals using DFS of the graph and so runs in O(V+E) (linear) time, which is asymptotically optimal because there is a matching lower bound).
Otherwise, if the graph is represented as an adjacency matrix, the algorithm requires $O(V^2)$ time.

3. Give a linear-time algorithm that takes as input a directed acyclic graph G and two vertices s and t and returns the number of simple paths from s to t in G.

The idea is to do Depth First Traversal of given directed graph. Start the traversal from source. Keep storing the visited vertices in an array say 'path[]'. If we reach the destination vertex, add path to "Result" array. Final, we count the length of the "Result" array, which represents the number of simple paths. The important thing is to mark current vertices in path[] as visited also, so that the traversal doesn't go in a cycle.

Pseudo-code:

```
1. Res ← empty list
2. Visited ← empty list
3. DFS(u, d, visited, path)
    visited[u] ← True
    path ← u
    If u and d are the same
        Res add path
    Else
    for each neighbor vertices v of u
        If v is unvisited
            DFS(v, d visited, path)
    path.pop
    visited u ← False
```

```
4. DFS(s,t,visited,path)
5. Return Length of Res
```

Time Complexity:

Depth-first search visits every vertex once and checks every edge in the graph once. Therefore, DFS complexity is O(V+E). As the above description , the time complexity for DFS to run on a graph is O(V+E) where V is number of vertices in the graph and E is number of edges in the graph. This algorithm is actually runs one time DFS so the time complexity is O(V+E), which is linear time.