CSCE 629: Analysis of Algorithms
Home Work - 4

Shabarish Kumar Rajendra Prasad
UIN - 228000166

November 2019

1 Question 1

The algorithm is shown below:

Algorithm: topological_sort $(G = \{V, E\})$

```
for each vertex v in V:
    indegree[v] = 0
for each edge (u,v) in E:
    indegree[v] += 1

Q = queue()
for each vertex v in V:
    if indegree[v] == 0:
        Q.enqueue(v)

i = 0
A = []
while (Q is not empty):
    u = Q.dequeue()
    A.append(u)
    i = i+1

    for each edge (u,v) in E:
        indegree[v] = indegree[v]-1

        if indegree[v] == 0:
            Q.enqueue(v)

if i < |V|:
    return "Cycle found"
else:
    return A
```

We first correctly assign the indegree of each vertex in the graph and insert all the vertices with indegree 0 into a First-in-First-out queue. Then for each element in the queue, we append them to the Array A, and for all its neighbors, we decrement the indegree which is the equivalent of deleting that edge. And if the indegree of any such vertex becomes zero, we add them to the queue. Finally, if the number of vertices added to the array is not equal to the number of vertices in the graph, we return that the graph contains cycles, or we return the array otherwise.

Correctness Proof

The correctness of the algorithm can be easily proved. A vertex enters the queue only if its indegree is 0. And every vertex that enters the queue first is first appended to the array A which is to be returned. Thus for two elements v and u, if u enters the array A at $A[s]$ and if v enters A at $A[t]$, and if $s < t$, then there could be no edge from v to u. This applies for any two vertices, that there will be no edge from $A[t]$ to $A[s]$, if $s < t$.

On the other hand, if the length of the array to be returned is less than the cardinality of the vertices, then there is a subgraph in the graph G, and in the subgraph, no vertices have the indegree 0. This should mean that the subgraph should contain cycles. Thus we just return that the graph contains cycle.

Time Complexity

The first for loop takes time equivalent to the cardinality of the vertices, which is in the order of $O(|V|)$. The second for loop takes time equivalent to the cardinality of the edges in the graph, which is in the order of $O(|E|)$. The third for loop like the first loop, takes time $O(|V|)$. The the while takes time in the order of $O(|V| + |E|)$. Thus the complexity of the entire algorithm can be described as $O(|V| + |E|)$.

2 Question 2

Consider that each vertex in the graph is labelled as a unique integer. The algorithm is shown below:

Algorithm: Input:$(G = \{V, E\})$

```
scc = strongly_connected_components(G)

G1 = new graph()
for each vertex v in V:
    for each edge (u,v) in E:
        if scc[u] != scc[v]:
            G1.insert_edge(scc[u],scc[v])

G1 = remove_multiple_edges(G1)
return G1
```

The first step of the algorithm is to use the algorithm taught in class to find the strongly connected components in the graph. We retrieve the strongly connected components in an array. Next, for every edge, we check if the edge belongs two distinct components in the scc array. If yes, we add a edge between two components in the scc array in a new graph. Then we use a linear time multiple edge removal algorithm, like the one we designed in Homework-3 and return the new graph.

Time Complexity

Finding the strongly connected components in the graph take time in the order of $O(|V| + |E|)$. The for loop also take time in the order of $O(|V| + |E|)$. Removing multiple edges also take time $O(|V| + |E|)$. Thus the time complexity of the entire algorithm can be explained as $O(|V| + |E|)$.

3 Question 3

The algorithm is given below:

Algorithm: Input: G={V,E}

```
SV = topological_sort(G(V))

for each v in V:
    cn[v] = 0

for j=0 to j=|V|-1:
    if SV[j] == s:
        break
cn[s] = 1

for i=j to i=|V|-1:
    if SV[i] == t:
        return cn[t]
    for each edge (SV[i],v) in E:
        cn[v] = cn[SV[i]] +cn[v]
```

The first step of the algorithm is to sort all the vertices of the graph in topological order as we did in Question 1. Then we shall compute the number of simple paths from the node s by using the formula:

$$cn[SV[i]] = \sum_{j<i \text{ and } (SV[i],SV[j])\in E} cn[SV[j]]$$

i.e., that is for two different j_1 and j_2 with $j_1 < i$ and $j_2 < i$, if $(SV[j_1], SV[i])$ and $(SV[j_2], SV[i])$ are two edges in E, and if paths from s to j_1 and a path from s to j_2 both exist, then two different paths from s to i can be drawn, one through j_1 and another through j_2. Based on this observation, the algorithm performs dynamic programming to find the total number of simple paths from s to t.

Time Complexity

The topological sorting takes a time in the order of $O(|V| + |E|)$ as shown in Question 1. The second for loop may take time in the order of $O(|V|)$ and the third for loop may take time in the order of $O(|E|)$. The total time complexity of the algorithm can be described as $O(|V| + |E|)$.