CSCE-629 Analysis of Algorithms

Spring 2019

Instructor: Dr. Jianer Chen
Office: HRBB 315C
Phone: 845-4259
Email: chen@cse.tamu.edu
Office Hours: T,Th 11:00 am–12:30 pm

Teaching Assistant: Qin Huang
Office: HRBB 315A
Phone: 402-6216
Email: huangqin@email.tamu.edu
Office Hours: MWF 3:00 pm–4:00 pm

Solutions to Assignment #4

(Prepared with TA Qin Huang)

1. Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Develop an $O(|V| + |E|)$-time algorithm using this approach. Your algorithm should also be able to tell when the input graph has cycles.

Solutions. Denote by $[u, v]$ the directed edge from vertex u to vertex v. Let Q be a First-In-First-Out queue that keeps all the vertices with degree 0. The output of the algorithm is given in an array A, in the topologically sorted order.

Algorithm 1 Pseudocode for Problem 1

Input: $G = (V, E)$

```
1:  for (each vertex u in V) do
2:      indeg[u] = 0;
3:  for (each edge [u, v] in E) do
4:      indeg[v]++;
5:  for (each vertex u in V) do
6:      if indeg[u] == 0 then
7:          Q ← u;
8:  p = −1;
9:  while (Q is not empty) do
10:     u ← Q; A[++p] = u;
11:     for (each edge [u, v] in E) do
12:         indeg[v]−−;
13:         if indeg[v] == 0 then
14:             Q ← v;
15: if p < n − 1 then
16:     return ("G has cycles.");
17: else
18:     return (A[0...n − 1]).
```

The array A maintains a partial topological order as vertices of in-dgree 0 are added in. Each vertex u is pushed into the queue Q at most once, either in step 7 if its indegree is 0 in the original graph G, or in step 14 when its indegree becomes 0 from 1. Therefore, the **While**-loop in steps 9-14 takes time $O(|V| + |E|)$. It is easy to verify that all other steps of the algorithm take time $O(|V| + |E|)$. Therefore, the time complexity of the algorithm is $O(|V| + |E|)$.

The correctness of the algorithm is easy to see. When we place a vertex u in the array $A[p]$, the indegree of u is 0, which means that all coming-in edges to u are from array elements $A[q]$ with $q < p$. Thus, if the algorithm returns at step 18, where all vertices of G are placed in the array A, then there will be no edge in G that goes from $A[q]$ to $A[p]$ with $p < q$, i.e., the array A gives a topologically sorted order for the vertices of the graph G.

On the other hand, if the algorithm returns at step 16, then the graph G has a subgraph G' in which no vertices have indegree 0. Then the subgraph G' must have cycles. This can be seen as follows. Start from any vertex in the subgraph G' and traverse G' following the reversed edge directions. Since each vertex in G' has indegree larger than 0, this traversing can never stop. Thus, the traversing must eventually repeat vertices, i.e., the subgraph G' must have cycles. Thus, if the algorithm returns at step 16, then the graph G must have cycles.

2. Let G be a directed graph with strongly connected components C_1, C_2, ..., C_k. The *component graph* G^c for G is a directed graph of k vertices w_1, w_2, ..., w_k such that there is an edge from w_i to w_j in G^c if and only if there is an edge from some vertex in C_i to some vertex in C_j. Develop an $O(|V|+|E|)$-time algorithm that on a given directed graph $G = (V, E)$ produces the component graph G^c for G. Make sure that there is at most one edge between two vertices in the component graph G^c.

Solutions. Denote by $[u, v]$ the edge from vertex u to vertex v. Use the algorithm given in class to construct all strongly connected components C_1, C_2, ..., C_k of G, which takes time $O(|V| + |E|)$. Assume that the result is stored in an array $scc[0..n-1]$ with $0 \le scc[v] \le k-1$ for all vertices v in G, such that two vertices u and v in G are in the same strongly connected component if and only if $scc[u] = scc[v]$. Let $CG[0..k-1]$ be an adjacency list for the component graph G^c, which, obviously contains at most $|V|$ vertices and $|E|$ edges.

Algorithm 2 Pseudocode for Problem 2

Input: $G = (V, E)$

```
1: use the SCC algorithm given in class to construct the array scc[0..n − 1];
2: for (each vertex u in V) do
3:     for (each edge [u, v] in E) do
4:         if (scc[u] ≠ scc[v]) then
5:             add scc[v] into the linked list CG[scc[u]];
6: use the algorithm given in Problem 3 in Homework 3 to remove repetitive edges;
7: return CG[0..k − 1];
```

As discussed in class, step 1 of the algorithm takes time $O(|V|+|E|)$. Steps 2-5 obviously take time $O(|V|+|E|)$. Finally, by what we have seen in Homework 3, step 6 takes time $O(|V|+|E|)$. In conclusion, the running time of the algorithm is $O(|V| + |E|)$. □

3. Given 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. Your algorithm needs only to count the simple paths, not list them. Note that different paths from s to t may share common vertices.

Solutions. First topologically sort the vertices of the graph $G = (V, E)$, and place them in an array $T[0..n-1]$. This takes time $O(|V| + |E|)$. Then for each vertex v of G, compute the number $cn[v]$ of simple paths from s to v. Since all edges can only go from left to right in the array $T[0..n-1]$ (i.e., all edges are of the form $[T[j], T[i]]$ with $j < i$), we have the following formula:

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

i.e., for two different j_1 and j_2 with $j_1 < i$ and $j_2 < i$, where $[T[j_1], T[i]]$ and $[T[j_2], T[i]]$ are edges in E, a path from s to $T[j_1]$ plus the edge $[T[j_1], T[i]]$ and a path from s to $T[j_2]$ plus the edge $[T[j_2], T[i]]$ give two different paths from s to $T[i]$. Based on this formula, a dynamic programming algorithm can be derived as follows.

Algorithm 3 Pseudocode for Problem 3

```
1: topologically sort the vertices of G and place them in an array T[0..n-1];
2: for (each v in V) do cn[v] = 0;
3: let j be the index such that T[j] = s, cn[s] = 1;
4: for (i = j + 1; i < n; i++) do
5:     u = T[i];
6:     if (u == t) then
7:         return (cn[t]);
8:     for (each edge [u, v] in E) do
9:         cn[v] = cn[u] + cn[v].
```

Since each edge is visited at most once in step 9, the total running time of the **for**-loop of steps 4-9 is $O(|E|)$. Therefore, the algorithm runs in time $O(|V| + |E|)$.