

CS41 Homework 3

This homework is due at 11:59PM on Friday, October 9. However, you may submit it up to two days late, by 11:59PM on Sunday, October 11 with no penalty (and without using any late days).

Write your solution using L^AT_EX. Submit this homework in a file named `hw03.tex` using [github](#).

It's ok to discuss approaches at a high level. In fact, I encourage you to discuss general strategies. However, you should not reveal specific details of a solution, nor should you show your written solution to anyone else.

The main **learning goals** of this homework are to practice algorithm design analysis/proof skills and for graph algorithms.

1. **DFS Running Time.** Consider the recursive version of DFS:

DFS(G, u)

- 1 Mark u as “Explored”
- 2 **for** each neighbor v of u
- 3 **if** v is not marked as “Explored”
- 4 Recursively invoke DFS(G, v)

Suppose this is implemented with G represented as an *adjacency list*. Specifically, the vertices of G are labeled with numbers $1, 2, \dots, n$ and for each vertex $i \in \{1, 2, \dots, n\}$, we have a list `adj[i]` containing the neighbors of vertex i (i.e., some number j appears in `adj[i]` whenever vertex j is a neighbor of i). Argue in a few sentences that DFS runs in time $O(m + n)$.

2. **Topological Ordering.** Recall that a *directed acyclic graph* (DAG) has a topological ordering that can be found using the following recursive algorithm (recall also that a *source vertex* is a vertex with *in-degree* 0, i.e., with no incoming edges):

TOPO-SORT(G)

- 1 **if** G is empty, **return** an empty list
- 2 Find a source vertex $v \in V$
- 3 Let $G' = (V', E')$ be the graph that results from removing v and all of its outgoing edges from G .
- 4 Recursively compute a topological ordering on G' (i.e., run TOPO-SORT(G'))
- 5 **return** v followed by the topological ordering returned by TOPO-SORT(G').

- (a) **Correctness.** For the above algorithm to be correct, we need to ensure that G has a source vertex (since the algorithm must find one). Prove that every DAG has at least one source vertex.

- (b) **Improved Running Time.** Suppose G is given as an adjacency list, where both incoming and outgoing edges are provided. That is, for a vertex v , we have a list `in[v]` of vertices u for which $(u, v) \in E$ (i.e., there is an edge from u to v) and a list `out[v]` of vertices u for which $(v, u) \in E$ (i.e., there is an edge from v to u).

In the most basic implementation of the topo-sort algorithm, finding a source vertex takes $\Theta(n)$ time since we look at all vertices until we find one with no incoming edges. This leads to a total running time of $\Theta(n^2)$.

We can do better. To start, we can convert the recursive algorithm to an iterative one:

ITER-TOPO-SORT(G)

```
1  Let  $L = []$  be an empty list //  $L$  will be our topological ordering
2  while  $G$  is not empty
3      Let  $v$  be a source vertex in  $G$ 
4      Append  $v$  to  $L$ 
5      Remove  $v$  and its outgoing edges from  $G$ 
6  return  $L$ 
```

Show how to modify ITER-TOPO-SORT so that it runs in time $O(m+n)$. Give pseudocode for your modified algorithm, and prove that its running time is indeed $O(m+n)$. As in the previous part, you may assume G is given as an adjacency list with both incoming and outgoing edges.

Hint: Maintain a list S of all known source vertices. After deleting a vertex, how can we update S efficiently (which vertices might become sources)?

- (c) **Non-DAGs.** Recall that the algorithm relies on the fact in part (a), which assumes that G is a DAG. Using the fact from part (a), argue that if at any point in the execution of ITER-TOPO-SORT, line 3 cannot find a source vertex (that is, when the remaining graph is non-empty but has no source vertex), then the original graph is not a DAG. Briefly describe how ITER-TOPO-SORT can be modified to produce an algorithm that detects whether a graph G has a directed cycle.

3. **Ethnographers.** (Kleinberg and Tardos, 3.12) You're helping a group of ethnographers analyze some oral history data they've collected by interviewing members of a village to learn about the lives of people who have lived there over the past two hundred years.

From these interviews, they've learned about a set of n people (all now deceased), whom we'll denote $P_1, P_2, \dots, P_n$. They've also collected facts about when these people lived relative to one another. Each fact has one of the following two forms:

- for some i and j , person P_i died before person P_j was born; or
- for some i and j , the lifespans of P_i and P_j overlapped at least partially.

Naturally, the ethnographers are not sure that all these facts are correct; memories are not very good, and a lot of this was passed down by word of mouth. So what they'd like you to determine is whether the data they've collected is at least *internally consistent*, in the sense that there could have existed a set of people for which all the facts they've learned simultaneously hold.

Give an efficient algorithm to do this: either it should propose dates of birth and death for each of the n people so that all the facts hold true, or it should report (correctly) that no such dates can exist—that is, the facts collected by the ethnographers are not internally consistent.