CS 330 – Summer 2019 – Lab 3 Solutions

Question 1. *Topological sort*

Find two different valid topological sorts of the following graph G:

Is G a DAG (directed acyclic graph)? How can you tell? Also, what is the time complexity of topological sort?

Solution

Algorithm: We will try to find a topological ordering of G. If we can, then we know that G is a DAG and we return True. If we cannot find a topological ordering, then we know that G is not a DAG and we will return False. To devise a topological ordering, we follow the algorithm described on p. 103-104 of Kleinberg and Tardos:
We declare a node to be âactive" if it has not yet been deleted by the algorithm, and we explicitly maintain two things:

a) for each node w, the number of incoming edges that w has from active nodes; and

b) the set S of all active nodes in G that have no incoming edges from other active nodes.

At the start, all nodes are active, so we can initialize (a) and (b) with a single pass through the nodes and edges. Then, each iteration consists of selecting a node v from the set S and deleting it. After deleting v, we go through all nodes w to which v had an edge, and subtract one from the number of active incoming edges that we are maintaining for w. If this causes the number of active incoming edges to w to drop to zero, then we add w to the set S. Proceeding in this way, we keep track of nodes that are eligible for deletion at all times, while spending constant work per edge over the course of the whole algorithm.
If we are able to add every node in G to our topological sort, we return True. Otherwise, we return False.
Equivalence Proof: To prove that we can identify a DAG using a topological sort, we need to prove that G is a DAG if and only if it has a topological odering.
Let's start by proving G is a DAG if it has a topological ordering. For this, we copy proof 3.18 from p. 101 of Kleinberg and Tardos:
Suppose, by way of contradiction, that G has a topological ordering $v_1, v_2, \ldots, v_n$, and also has a cycle C . Let v_i be the lowest-indexed node on C, and let v_j be the node on C just before v_i - thus (v_j, v_i) is an edge. But by our choice of i, we have $j > i$, which contradicts the assumption that $v_1, v_2, \ldots, v_n$ was a topological ordering.
Now, we may use a similar argument to prove that G is not a DAG if it does not have a topological ordering. If G has a cycle C, then there is a pair of vertices $a, b \in C$ such that it is possible to find a path from a to b and a path from b to a. If we try to sort G topologically, we fail if we put a before b in the sort, and also if we put b before a.
Correctness: As proven in class, a directed cycle cannot have a topological ordering, because a directed cycle has no root nodes. Since this algorithm terminates once there are no root nodes, it must eventually stop if the input contains a directed cycle. Conversely, when the input does not contain a directed cycle, it produces a topological ordering of the nodes.
Time complexity: Creating the topolgical order takes $O(m + n)$ time. Firstly, we construct our sets (a) and (b) with a single pass through all the vertices and edges, which takes $O(m + n)$. Then, we visit each node once, which takes $O(m)$ time. The number of updates we make to (a) and (b) at each node v corresponds to the number of outgoing edges at v, so the total number of all updates made at every node is $O(n)$. Taken together, it takes $O(m + n)$ time.

Question 2. *Job scheduling*

IS&T services at BU has some shared resources available for students and faculty. To make use of these resources each person has to submit the exact time that she intends to run her job and how long it will take. Then IS&T will decide to allow some of the jobs to run but will reject others. Their goal is to fulfill **as many requests as possible** with the caveat that **only one job at a time** can run. (Jobs can only be started at the exact time that was requested, otherwise they are rejected.)

In the following, you will devise an algorithm to decide which jobs to run or reject to maximize the number of jobs that are accepted. We say that a schedule is *valid* if it has no overlapping jobs. The schedule is *optimal* if it has the maximum number of jobs.

a) Assume the requests are given as the tuples (start time, duration in mins) below. Which jobs should be included in an optimal schedule? Note, that there is more than one optimal schedule.

- job_1 (9:00, 20 m)
- job_2 (9:15, 10 m)
- job_3 (8:00, 65 m)

- job_4 (10:00, 120 m)
- job_5 (9:30, 40 m)
- job_6 (10:15, 60 m)
- job_7 (9:55, 70 m)
- job_8 (10:00, 12 m)
- job_9 (10:00, 10 m)

b) Compare the two optimal solutions that you found. Find a pair of jobs in the two schedules, so that you can exchange one for the other and still have a valid schedule. Why were you able to make the exchange? Try to formulate why it is always true, that if you have more than one optimal schedule, then you can always find such a pair of jobs to switch between two optimal schedules.

Solution

a) Two optimal solutions are:

(a) job_3, job_2, job_5, job_6

(b) job_3, job_2, job_8, job_6

b) You can switch job_5 and job_8. The optimal algorithm for this scheduling problem is to order jobs by increasing finishing time. Then, in greedy fashion, always pick the next finisher that is compatible with the ones already in the schedule. Call the schedule found by the greedy algorithm A. Now suppose there is another optimal schedule besides the one found by the greedy algorithm, and call it B. At some point, the schedules must be different, so assume both schedules are identical on the first $i-1$ jobs, but differ on the i^{th}. By definition, the i^{th} job of schedule A has an earlier end time than the i^{th} job of schedule B, so the i^{th} job in schedule A is compatible with job $i+1$ in schedule B. And since job $i-1$ is the same in both schedules A and B, the i^{th} job in schedule A is compatible with job $i-1$ in schedule B, and we may switch this job in to schedule B.

Question 3. *Addapted from Chapter 3, Exercise 11, on p. 111-112 (additional examples in text).*

There are n computers in a system, labeled C_1, C_2,..., C_n, and as input you're given a collection of trace data indicating the times at which pairs of computers communicated. The data is a sequence of triples (C_i, C_j, t_k); such a triple indicates that C_i and C_j exchanged bits at time t_k. There are m triples total. The triples are presented in sorted order of time. If the virus was inserted into computer C_a at time x, could it possibly have infected computer C_b by time y?

Design an algorithm such that, given a collection of trace data, the algorithm should decide whether a virus introduced at computer C_a at time x could have infected computer C_b by time y. The algorithm should run in time $O(m+n)$.

Solution

Graph construction: Create a directed graph, G, with nodes representing each computer at an instant in time when it is communicating (in other words, if there are multiple triplets that contain C_x, create a node for each distinct time stamp they correspond with another machine). If we have n time-stamped events, this produces $2n$ nodes. Now create two types of edges: communication edges and edges going forward in time. Start by going through the triplets (x, y, t), creating a directed edge from the node of the first computer in the triplet C_x at time t to the other computer in the triplet represented by its respective node C_y at time t, and an edge in the opposite direction. This creates $2n$ communication edges (two per triplet). Additionally, create directed edges to the node that represent the same computer at the next later time stamp (if it exists). For example, if there are nodes represents C_x at time 4, 22, and 36, add a directed edge from $(C_x, 4)$ to $(C_x, 22)$ and from $(C_x, 22)$ to $(C_x, 36)$. This creates at most $2n$ more forward-in-time edges (at most one per node).

Algorithm: Now perform a BFS (keeping the directed edges in mind) from the node representing (C_a, t) whose time stamp t is $\geq$ to the given time of infection x, and return true and the path if a node representing C_b prior to time y is discovered. The implementation of BFS takes O$(m+n)$ time. In our construction, $m \leq 4n$, so this is O(n).

Correctness : To prove that this algorithm is correct, we need to show that: if an infection can occur then BFS identifies a path, **and** if an infection did not occur, BFS does not identify a path. In other words, C_b can be infected if and only if the BFS returns true. (Note that proving only one direction of the iff is not enough: the algorithm must answer correctly whether or not there was an infection sequence.)

From the problem statement, an infection can arise if and only if there is a time-increasing sequence of communications between computers $(a, x, t_1), (x, y, t_2), ...(z, b, t_k)$, where $t_1 \leq t_2 \ldots \leq t_k$. By our graph construction, if there is such an infection sequence, we will have a sequence of edges in our graph forming a directed path: $(a, t_1) \rightarrow (x, t_1)$, $(x, t_1) \rightarrow (x, t_2), (x, t_2) \rightarrow (y, t_2), \ldots, (z, t_k) \rightarrow (b, t_k)$. This path will be identified by BFS, so infection sequences lead to BFS outputting yes.

To prove the other direction: no infection sequence implies BFS outputs no, it is easiest to prove the contrapositive: BFS outputs yes implies infection sequence. So suppose BFS outputs yes. Then there exists some sequence of edges $(a, t_1) \rightarrow (a, t_2), \ldots (x, t_k) \rightarrow (b, t_k)$ that BFS identified. By our graph construction, we can trace an infection along this path: a forward-in-time edge indicates that an infected machine stays infected over time, and a communication edge indicates that an infected machine infects another machine. Equivalently, since the BFS path is time-increasing, we can recover a time-increasing sequence of communications between computers $(a, x, t_2), (x, y, t_8), ...(z, b, t_k)$ from the BFS path, which is an infection sequence.