CS3510-A: Design and Analysis of Algorithms, Spring 2021

Homework-3

February 2, 2021

DUE DATE: Tuesday, February 9, 11:59pm

Note-1: Your homework solutions should be electronically formatted as a single PDF document that you will upload on Gradescope. If you have to include some handwritten parts, please make sure that they are very clearly written and that you include them as high resolution images.
Note-2: Please think twice before you copy a solution from another student or resource (book, web site, etc). It is not worth the risk and embarrassment.
Note-3: You need to **explain/justify** your answers. Do not expect full credit if you just state the correct answer.
Note-4: You will get 2 extra points if you submit electronically typed solutions instead of hand-written.

Problem-1 (30 points)

You are given a DAG. How would you check in linear time if the DAG includes a Hamiltonian path, i.e., a (directed) path that traverses every vertex exactly once?

Note 1: Please provide a detailed description of your solution in plain English or in pseudocode.
Note 2: You are also allowed to use BFS and DFS as blackbox algorithms (you do not need to explain how it works if you use it) so long as you detail the inputs and outputs to the algorithms as well as any modifications you make.

Solution

- Find all source nodes (This can be done in O(n))
- If there are multiple source nodes, a Hamiltonian path is not possible
- Perform a DFS from the source node over the graph. DFS process will have to be modified to allow exploring nodes that have already been explored in previous path branches, but not the current path's nodes (this way all possible paths from the start are explored).
- If the longest path length during the traversal is equal to the number of nodes in the graph, then a Hamiltonian path is possible. (This step can be done in $O(n+m)$, the same a normal DFS)

Runtime:
The overall process involves: $O(n) + O(n+m)$.
Therefore the overall runtime for the process should be: $O(n+m)$

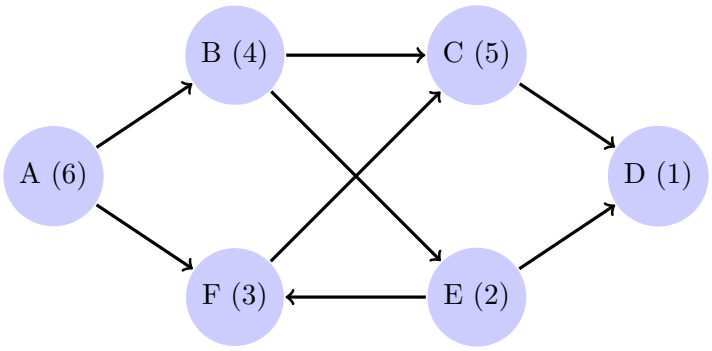
Explanation:
By finding a source node to start from, if the DFS is able to find a path which has a depth that extends through the full graph and only visits each node once during the path, the path is by definition a Hamiltonian path. If there are more than one source node, then starting a path from one of the source nodes requires that the remaining source nodes can not possibly be visited by a path starting from any other node.

Problem-2 (35 points)

You are given a directed graph in which each node $u \in V$ has an associated capacity c_u which is a positive integer. Define the array *bottleneck* as follows: for each $u \in V$,

$$bottleneck[u] = \text{capacity of the lowest-capacity node reachable from } u \text{ (including } u \text{ itself).}$$

For instance, in the graph below (with capacities shown inside each vertex), the bottleneck values of all nodes are equal to 1. Your goal is to design an algorithm that fills in the entire bottleneck array (i.e., for all vertices).



(a) Design a linear-time algorithm that works for DAGs.

(b) Extend your previous answer to a linear-time algorithm that works for all directed graphs.

Note 1: Please provide a detailed description of your solution in plain English or in pseudocode.

Note 2: You are also allowed to use BFS and DFS as blackbox algorithms (you do not need to explain how it works if you use it) so long as you detail the inputs and outputs to the algorithms as well as any modifications you make.

Solution

Process:

- Build a set of all sink nodes in the graph (S). (This can be done in O(n))
  - Iterate over all nodes in the graph.
  - Add any nodes with no edges that originate from them to the set of sing nodes (S).
- Perform a traversal (similar to DFS) from each node in the set S over the graph (This can be done in the normal O() for DFS, O(n+m)):

Data: s: starting node for traversal
v_s: value of the starting node
v: value to use to update other nodes during traversals (initially as v_s)

Process:

- Perform DFS starting from s
- $v = v_s$ (the value that will be used to update node values should be the value of the sink node)
- For each node (n) explored during the traversal:
  - if value of current node (v_n) less than v, then set $v = v_n$ and proceed
  - else, set the value of current node to v ($v_n = v$)

Runtime:
The overall process involves: $O(n) + xO(n + m)$
Where x is the number of sink nodes. There will at most be n sink nodes, so the overall runtime should be: $O(n(n + m))$

Explanation:
By finding all sink nodes the algorithm identifies nodes that can have no bottleneck value besides their own value. By traversing from each sink node the algorithm ensures that each sink nodes parent node has the bottleneck value for the smallest bottleneck "downstream" from it. By changing the update value from the sink node's value to the minimum value found along the traversal, the algorithm guarantees that any bottlenecks that don't originate from sink nodes are still applied to their relevant parent nodes.

Problem-3 (35 points)

You're helping some security analysts monitor a collection of networked computers, tracking the spread of an online virus. There are n computers in the 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. Thus the data is a sequence of ordered 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.

We'll assume that the triples are presented to you in sorted order of time. For purposes of simplicity, we'll assume that each pair of computers communicates at most once during the interval you're observing.

The security analysts you're working with would like to be able to answer questions of the following form: If the virus was inserted into computer C_a at time x, could it possibly have infected computer C_b by time y? The mechanics of infection are simple: if an infected computer C_i communicates with an uninfected computer C_j at time t_k (in other words, if one of the triples (C_i, C_j, t_k) or (C_j, C_i, t_k) appears in the trace data), then computer C_j becomes infected as well, starting at time t_k. Infection can thus spread from one machine to another across a sequence of communications, provided that no step in this sequence involves a move backward in time. Thus, for example, if C_i is infected by time t_k, and the trace data contains triples (C_i, C_j, t_k) and (C_j, C_q, t_r), where $t_k < t_r$, then C_q will become infected via C_j. (Note that it is okay for t_k to be equal to t_r; this would mean that C_j had open connections to both C_i and C_q at the same time, and so a virus could move from C_i to C_q.)

For example, suppose $n = 4$, the trace data consists of the triples:

$$(C1, C2, 4), (C2, C4, 8), (C3, C4, 8), (C1, C4, 12)$$

and the virus was inserted into computer C_1 at time 2. Then C_3 would be infected at time 8 by a sequence of three steps: first C_2 becomes infected at time 4, then C_4 gets the virus from C_2 at time 8, and then C_3 gets the virus from C_4 at time 8. On the other hand, if the trace data were:

$$(C2, C3, 8), (C1, C4, 12), (C1, C2, 14)$$

and again the virus was inserted into computer C_1 at time 2, then C_3 would not become infected during the period of observation: although C_2 becomes infected at time 14, we see that C_3 only communicates with C_2 before C_2 was infected. There is no sequence of communications moving forward in time by which the virus could get from C_1 to C_3 in this second example.

Design an algorithm that answers questions of this type: 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 $O(m + n)$.

Note 1: Please provide a detailed description of your solution in plain English or in pseudocode. Also, explain the time complexity and correctness of the prescribed algorithm.

Note 2: You are also allowed to use BFS and DFS as blackbox algorithms (you do not need to explain how it works if you use it) so long as you detail the inputs and outputs to the algorithms as well as any modifications you make.

Solution

Process:

- Use communication data to build an undirected weighted graph, where the time of communication is used as the weight for the connection. (This can be done in $O(n+m)$)

- Perform a DFS on the built graph from C_a to C_b, but only explore paths with weights (w) where $x \leq w \leq y$ and using paths with weights $\geq$ the weight from the previous steps. (This can be done in $O(n+m)$, the same as normal DFS)

- If the path is not possible with connections where $x \leq weight \leq y$ and with steps ascending in weight, then C_b won't be infected.

Runtime:
Because the process uses two steps with $O(n+m)$ the overall process will have $O(n+m)$.

Explanation:
By building out a graph of all communication histories, if there is any path of infection between the two computers it will appear as some path in the graph. By traversing on connections with times/weights between x and y only communications after an infection was introduced to the system and before the time of inquiry are used (alternatively this step could be replaced by only building the graph with communications that occur between the two times). By starting at the C_a node and only traversing on connections by ascending order, a path will only be considered if it involves the transmission of a virus from a previous node. By this criteria any path of connections that stem from the C_a node, step in chronological order after time x, occur before the time of inquiry (y), and reach the C_b node will result in the infection of the C_b node.