CS-GY 6033: Homework #5

Due on November 27, 2023

Professor Yi-Jen Chiang

Runze Li
Nxxxxxxxx
rl50xx@nyu.edu

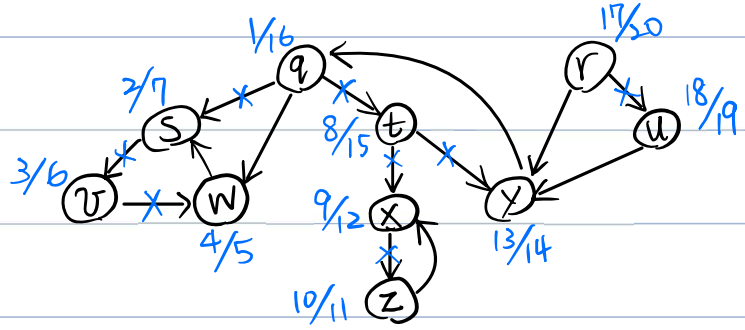
Tzu-Yi Chang
Nxxxxxxxx
tc39xx@nyu.edu

Jiayi Li
Nxxxxxxxx
jl156xx@nyu.edu

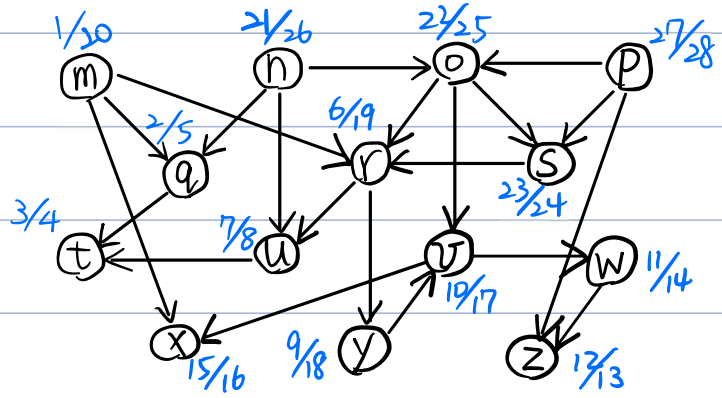
November 27, 2023

1.

(a)

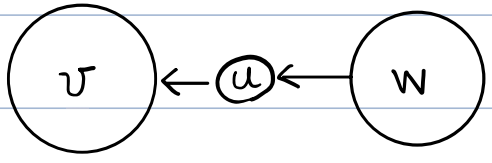


(b)



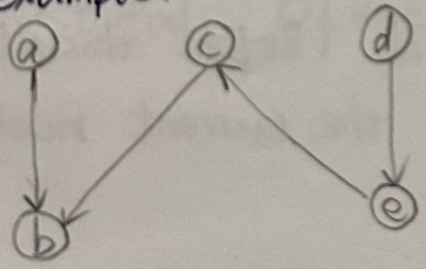
P n o s m r y v x w z u q t

2.



In this directed graph G, a vertex u has both incoming and outgoing edges. If all of the vertices in v are visited first, they form their depth-first-search tree(s). If u is chosen to be visited next, then it forms its own depth-first-search tree containing only u since it has no edges to visit.

Here is the example:



Apply DFS on the graph. Start with a and get stuck in b. Start again with c, as b is finished, then get stuck in c. Now, vertex c, with incoming edge from e and outcoming edge to b, ends up in a dfs tree containing only c.

3.

(a) def ComputeInDegrees(V, E, adjacency-list):

Creat an empty array inDegree with size |V|, initialize values in array to 0.

for each vertex u in adjacency-list:

O(V+E) { for each vertex v in the adjacency-list of u:

{ inDegree[v] increments by 1. // v has an incoming edge from u

return inDegree

We visit each vertex in O(V) and each time some edges, which add up to a total of O(E), resulting in the worst-case running time of O(V+E).

(b) def TopologicalSort(V, E, adjacency-list, inDegree array):

O(V) Create an empty queue, add vertices with in-degree 0 into the queue.

Create an empty array res, where we'll save the result of this topological sorting.

while queue is not empty:

{ u = vertex popped from queue.

Add u to res (or output it).

O(V+E) for each vertex v in the adjacency-list of u:

{ inDegree[v] decrements by 1.

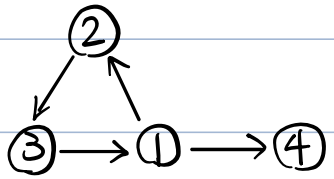
if inDegree[v] is 0:

{ Push v into queue.

In this topological sorting algorithm, we initialize the queue with only O(V) running time, and then a total worst-case time O(V+E) is performed since we visit each vertex (|V|) and each edge (|E|) once in the while loop.

(c) If G contains cycles, the topological sorting algorithm will fail, leaving several vertices (those in the cycles) that can not be sorted. Because the vertices all has incoming edges to form cycles, they will not be pushed into the queue by any chance. As we remove vertices with in-degree 0, the queue become empty and break the while loop, leaving the cycles that cannot be topologically sorted.

4.



In the first DFS, if we start from vertex 1, we get the finish time order as (3, 2, 4, 1).

As we perform the second DFS on the original graph in the order of increasing finish time, it turns out that all vertices are strongly connected since all remaining vertices are reachable from vertex 3. However, there should be two strongly connected components, which are (1,2,3) and (4). So the new algorithm is incorrect.

5. (a) Add a step to DFS. When finishing the search of a node and go back to its parent node, store this "return" traverse, which will be the second time to traverse that edge.

Analysis of time: same as DFS $O(V+E)$.

(b) There are 2 kinds of edge of undirected graph: tree edges and back edges.

Start with any of the nodes, after first visit of the white node, set the color of the node to be gray and assign the label "tree edge" for this edge

At each node, we go to the child node with white color recursively.

If there is no white child node of the current node, we will check the label of the edge to the gray child node

We will first go to the gray child node of the edge with empty label. Then go back to current node.

If there is no edge with empty label, we will go to the gray child node through the "tree edge". And set the color of current node to be black. (Before this step, there must be only one tree edge of the current node.)

Analysis of time complexity:

We will go through each edge twice. And we will call the algorithm on each vertex once.

Total is $O(V+E)$

6.

(a)

(i) ① Starting from an arbitrary vertex, we perform several DFS(s) on G to compute the finish time for each vertex, forming topological sorting order. O(V+E)

② Create a hash map M to maintain the corresponding vertex u for each vertex v in V, such that value for M(v) = vertex u with min{L(u)| there is a path from u to v in G}.

③ Iterate through the vertices in topological order. For each vertex u, update M(v) for each vertex v that has an incoming edge from u such that M(v) holds the corresponding vertex with the minimum reaching lable. M(v) = vertex with the min label {M(v), M(u) if exists, u}. O(V+E)

(ii) Since we have computed the map M, including the corresponding vertex of the minimum reaching label for each vertex in V, we simply iterate through all the vertices and calculate $D_L(u,v) = L(v) - L(M(v))$. If M(v) doesn't exist, then $D_L(u,v) = -\infty$. Compare the $D_L(u,v)$ among all the vertices in V and determine the vertices u^*, v^* with maximum $D_L(u^*,v^*)$. This can be done in O(V) time.

So the overall running time for part(i)+part(ii) is O(V+E) worst-case time.

(b) ① Now that G may contain cycles, first we need to find the strongly connected components.

O(V+E) a) Perform DFS on G to compute the finish time for each vertex.

b) Generate transpose graph G^T.

c) Perform DFS on G^T in the order of decreasing finish time from the first DFS.

O(V+E) Each DFS tree is viewed as a separate strongly connected component. Merge the vertices in the strongly connected component into a new vertex i, with $L(i) = \min\{L(v) \mid v \in SCC\}$. After merging, we will get a new DAG G'.

② Perform the algorithm provided in part (a)(i) on G' and get the

O(V+E) hash map M. In the general case, we initialize this min_reaching_map for some vertices inside the SCC(s) in part ① c). $M(v) =$ vertex i where $v \in SCC$ and $v \neq i$ and $L(i) = \min\{L(v) \mid v \in SCC\}$.

③ Iterate through all the vertices in V to compute $D_L(u,v) = L(v) - L(M(v))$

O(V) where $M(v) = M(i)$ for vertex i is the merged vertex for the SCC containing v. Compare to get the maximum $D_L(u^*, v^*)$ and the corresponding u^* and v^*.

Since DFS and finding SCC take O(V+E) time and finding the final u^*, v^* takes only O(V), the overall worst-case time is O(V+E).