

1.Exercise 22.1-4.

Given an adjacency-list representation of a multigraph $G = (V, E)$, describe an $O(V + E)$ -time algorithm to compute the adjacency-list representation of the "equivalent" undirected graph $G' = (V, E')$, where E' consists of the edges in E with all the multiple edges between two vertices replaced by a single edge with all the self-loops removed.

-Make an empty adjacency-list.

-If a vertex V_i appears in V_j 's list copy V_j to the new adjacency-list for V_i .

-Repeat previous step for all vertices.

-This will "flatten" the list into an undirected adjacency-list. These lists will have duplicates (due to multiple edges and loops in multigraphs).

-Create another empty adjacency-list. Create an empty vector.

-For each list check for duplicate entries by marking a vertex's existence in the vector. Copy the vector into the new adjacency-list for that vertex. Repeat for all vertices. So for each V_i in the adjacency list mark all it's attached vertices in the vector, copy the vector to the new adjacency-list, and repeat after clearing the vector.

2.Exercise 22.4-2.

Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t , and returns the number of paths from s to t in

G. For example, in the directed acyclic graph of Figure 22.8, there are exactly four paths from vertex p to vertex v : pov , $poryvm$ $posryv$, and $psryc$. (Your algorithm needs to count the paths, not list them.)

There are many methods to perform this operation in linear time. First we know that DFS is Big-Theta($V + E$). Topologically sort the vertices as DFS "finishes" them. This takes $O(1)$ time in addition to the DFS time. Now we have a linked list of vertices. We start at node s in this list and follow all possible pointers from s . Anytime our traversal path encounters t we increment a counter and start at the next possible path (might not be s since we must exhaustively search all links from each encountered node). We can see that this process will take at most Big-Theta($V + E$) since that is the maximum number of links in our list.

Alternatively we could alter the DFS algorithm slightly to achieve better performance for this specific need. We could have DFS start at s and stop whenever it hits t . To do this we would have to increment a counter every time we hit t in our traversal. Please note that this algorithm is not complete as written so don't take off points. It was just added because it seems more intuitive than using a full DFS and topological sort.

3. You are given two lists of n vertices. One list contains them in the order they were discovered during a DFS, and the other list contains them in the order they were finished. Describe an algorithm which computes the discovery time and finishing time of each vertex, given the two lists, or else determines that

the input data is bogus (i.e. didn't really come from a DFS). What are the time and space complexities of your algorithm?

This algorithm is divided into two components. First we compute the finishing time, then we use this to compute discovery time. We assume that it takes no time to traverse the tree, only time passes when coloring nodes (changing state from white->gray->black).

Part 1:

- Create an 2-d array size $[2*n,2]$. The second row will track "discover" versus "finished". A 1 is finished a 0 is discovered.

- We locate the first node in the finished list in our discovery list and store it's index as *max*. We make the *cur* pointer the index of the first element in the finished list.

- Count the number of nodes from 0 to *max* in the discovery list. Then count the number of nodes from the the 0 to *cur* in the finished list. Add these two counts together to get a finished time. Place this node in the array at $[count,0]$ and mark $[count,1]$ as a 1 (finished).

- Increment the *cur* marker in the finished list and find it's node's index in the discovered list. If it's index exceeds *max* replace *max* with it's index in the discovered list.

- Repeat for all nodes in the finished list.

- The array now contains finished times for all nodes.

Part 2:

- Remove one node from the front of the discovered list

and insert it at the first empty spot in the array [empty,0] and set [empty,1] to 0 indicating discovered time.

-Repeat for all nodes in discovered list.

The array now contains a list of what happened at each time instance. We can search this array and ensure that no nodes are finished before their discovered to ensure the validity of the data.

The runtime complexity is $O(n \lg n)$ and the space complexity is $O(4 * n)$ since we are tracking finished/discovered in a poor manner. However space complexity can be enhanced by altering the storage method to storing data each in the same array element instead of it's own and we could achieve $O(n)$.

4. In one of the following two arrays, the top row is a list of nine vertices in discovery order, and the bottom row contains the same vertices in finishing order, for some DFS. (The other array contains bogus data.) Which is which, and why? Draw the DFS forest corresponding to the array containing the real data.

1	4	7	2	6	3	8	9	5
4	3	8	7	6	9	2	1	5

1	4	2	6	8	7	5	9	3
2	6	4	9	5	3	7	8	1

We could use the above algorithm to discover the faulty data. However, instinctively we see that in the first array 1 is discovered first. It should be listed as finished last however 5 finishes last. This can't happen in a DFS. The graph of the valid array (array 2) is drawn by hand below.