Assignment 2

CSCI 3110: Design and Analysis of Algorithms

Due May 29, 2018

Banner ID: ________________	Name: ________________________
Banner ID: ________________	Name: ________________________
Banner ID: ________________	Name: ________________________

Question 1 (20 marks) We discussed in class how to determine whether a graph G is connected, in $O(n+m)$ time. If G represents a computer network, connectivity is important but doesn't ensure reliability. What if a mouse gnaws through one of the wires? Can all computers still communicate with each other, that is, is the network still connected? Formally, we would like to answer the following question: Given a *connected* graph G, does G contain an edge e whose removal disconnects G, that is, an edge e such that the graph $G_e = (V, E \setminus \{e\})$ is disconnected? If the answer to this question is no, then the mouse has to nibble through at least two wires to disconnect the network; we call the graph 2-*edge-connected* in this case. (In general, a graph is k-edge connected if at least k edges need to be removed to disconnect the graph.)

It turns out that deciding whether a given graph is 2-edge-connected also takes $O(n+m)$ time, using a clever strategy based on depth-first search. Your task is much simpler: Develop an algorithm that takes $O(n \cdot m)$ time to decide whether a given graph is 2-edge connected. As you will come to expect as the norm in this course, there are three parts to a complete answer:

- Give a clear description of your algorithm in plain English (preferable) or pseudo-code (if the English description becomes too convoluted). (To check whether your description is clear, ask yourself whether a competent computer scientist would be able to implement your algorithm based solely on your description. The answer should be yes.) Your algorithm may use any tools we discussed in class as building blocks, as long as these tools do not directly solve the problem you are asked to solve. (Example: If I ask you to describe a sorting algorithm, "use Merge Sort" is not enough; you need to give the details of the algorithm. If I ask you to solve some optimization problem where the first step in your algorithm is to sort the input, then "sort the input using Merge Sort" is sufficient, without describing the inner workings of Merge Sort.)

- Prove that the algorithm is correct, that is, that it gives the desired answer for every valid input. (For some algorithms, as the one discussed here, this proof can be fairly simple without a need for much formal machinery, but it is required.)

- Prove that the algorithm achieves the desired running time for every possible input. (Again, this proof may be simple, but it is required.)

How to approach the problem: First observe that an $O(m^2)$-time algorithm is easy to obtain: For every edge e, the graph G_e can be computed from G in $O(n+m)$ time by making a copy of G and removing edge e. Since G is connected, we have $m \geq n-1$, that is, $O(n+m) = O(m)$. As discussed in class, the connected components of G_e can be computed in $O(n+m-1) = O(m)$ time. Thus, testing whether each graph G_e is connected takes $m \cdot O(m) = O(m^2)$ time. Since G is 2-edge-connected if and only if each graph G_e is connected, we thus have a valid $O(m^2)$ algorithm for testing whether G is 2-edge-connected.

To reduce the running time to $O(nm)$, you want to avoid testing for *every* edge whether G_e is connected. Specifically, you want to divide the edge set E into two subsets E_1 and $E_2 = E \setminus E_1$ such that $|E_1| \in O(n)$ and you can guarantee, without explicitly testing that this is true, that G_e is connected for every edge $e \in E_2$. Thus, you only have to test whether G_e is connected for every edge $e \in E_1$. Can you use a spanning tree of G to identify such a partition of E into two sets E_1 and E_2?

Question 2 (10 marks) Rooted trees are used extensively to represent ancestry relationships in various contexts. For example, concept hierarchies in computational linguistics or the evolutionary history of a set of species in computational biology are often represented as rooted trees. The simplest question an algorithm manipulating these trees can ask is whether a node u is an ancestor of another node v. Consider a tree T and assume that every node in the tree has a unique ID between 0 and $n-1$ (the tree has n nodes). Your task is to build a data structure which, given a pair of node IDs (u, v) decides whether the node with ID u is an ancestor of the node with ID v. Your algorithm should take $O(n)$ time to build the data structure. The size of the data structure should be $O(n)$. Answering an ancestry query for any given pair (u, v) using the constructed data structure should take constant time.

Just as in your answer to Question 1, give clear descriptions of the data structure, of the algorithm to construct it, and of the query procedure; prove that the query procedure gives the correct answer for every possible pair (u, v) of node IDs; and prove that the construction of the data structure and the query procedure take $O(n)$ and $O(1)$ time, respectively.

How to approach the problem: For every pair (u, v), there are three possibilities: u may be an ancestor of v, v may be an ancestor of u or neither u nor v is an ancestor of the other. Now compute a preorder numbering and a postorder numbering of the nodes in T. How do u and v's preorder and postorder numbers relate to each other in each of these scenarios? Can you construct the desired data structure based on your observations?