

---

**Collaboration Policy:** You are encouraged to collaborate with up to 3 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite by naming the book etc. or listing a website's URL. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

---

**Collaborators:** list collaborators's computing IDs

**Sources:** Cormen, et al, Introduction to Algorithms. (*add others here*)

---

### PROBLEM 1 *Proof about MSTs*

Let  $e = (u, v)$  be a minimum-weight edge in an undirected connected graph  $G$ . Prove that  $e = (u, v)$  belongs to some minimum spanning tree of  $G$ .

**Solution:** The definition of a MST is a tree of total minimum weight that connects all nodes in a graph. Since  $e$  is a minimum-weight edge in  $G$ , it must be in the MST in order for the MST to have a total minimum weight. If  $e$  is not included, then there are two cases:

In Case 1,  $e$  is replaced by an edge of equal weight  $e'$ , meaning  $e'$  edge is also a minimum-weight edge. This creates a new unique MST of  $G$ , but since there are multiple possible MST's for a given graph, there still exists a MST of  $G$  that includes  $e$ .

In Case 2,  $e$  is replaced by an edge of larger weight  $e'$ , which contradicts the definition of a MST. Since the MST now includes  $e'$ , it is no longer a tree of total minimum weight since there exists an alternate lower-cost path. Therefore, this is not a valid MST and proves that a MST of  $G$  must include  $e$ .

### PROBLEM 2

An airline, Gamma Airlines, is analyzing their network of airport connections. They have a graph  $G = (A, E)$  that represents the set of airports  $A$  and their flight connections  $E$  between them. They define  $\text{hops}(a_i, a_k)$  to be the smallest number of flight connections between two airports. They define  $\text{maxHops}(a_i)$  to be the number of hops to the airport that is farthest from  $a_i$ , i.e.  $\text{maxHops}(a_i) = \max(\text{hops}(a_i, a_j)) \forall a_j \in A$ .

The airline wishes to define one or more of their airports to be "Core 1 airports." Each Core 1 airport  $a_i$  will have a value of  $\text{maxHops}(a_i)$  that is no larger than any other airport. You can think of the Core 1 airports as being "in the middle" of Gamma Airlines' airport network. The worst flight from a Core 1 airport (where "worst" means having a large number of connections) is the same or better than any other airport's worst flight connection (i.e. its  $\text{maxHops}()$  value).

They also define "Core 2 airports" to be the set of airports that have a  $\text{maxHops}()$  value that is just 1 more than that of the Core 1 airports. (Why do they care about all this? Delays at Core 1 or Core 2 airports may have big effects on the overall network performance.)

**Your problem:** Describe an algorithm that finds the set of Core 1 airports and the set of Core 2 airports. Give its time-complexity. The input is  $G = (A, E)$ , an undirected and unweighted graph, where  $e = (a_i, a_j) \in E$  means that there is a flight between  $a_i$  and  $a_j$ . Base your algorithm design on algorithms we have studied in this unit of the course.

**Solution:**

$\text{hops}(a_i, a_j)$  should be implemented with a modified BFS. The algorithm should stop as soon as  $a_j$

is found and return the distance from  $a_i$  to  $a_j$ .

$\text{maxHops}(a_i)$  should be implemented with BFS using  $a_i$  as the starting node. Return the distance of the longest path from  $a_i$ . BFS has a time complexity of  $\Theta(V + E)$ , therefore  $\text{maxHops}()$  has a time complexity of  $\Theta(A + E)$

To be a Core 1 airport, the airport must have the smallest  $\text{maxHops}()$  value. First, call  $\text{maxHops}(a_n)$  on every airport  $a_n$  in set  $A$  and store those values in a list,  $\text{maxList}$ . Additionally, store each airport  $a_n$  in a list,  $\text{airportsList}$  according to its  $\text{maxHops}$  value. For example, if  $\text{maxHops}(a_1) = 5$ , store  $a_1$  at  $\text{airportsList}[5]$ . This takes  $\Theta(A)$  since we must check every airport.

Sort  $\text{maxList}$  to determine the smallest  $\text{maxHops}()$  value. Store that value as  $x$ . All the airports at  $\text{airportsList}[x]$  are a Core 1 airport. All the airports at  $\text{airportsList}[x + 1]$  are a Core 2 airport.

The total time complexity is  $\Theta(A(A + E) + n \log n) = \Theta(A^2 + AE + n \log n) = \Theta((A^2 + A^3 + n \log n) = \Theta(A^3)$  since we must call  $\text{maxHops}()$  on every single airport, then sort  $\text{maxList}$  to find the smallest value.

### PROBLEM 3 Vulnerable Network Nodes

Your security team has a model of nodes  $v_i$  in your network where the relationship  $d(v_i, v_j)$  defines if  $v_j$  depends on  $v_i$ . That is, if  $d(v_i, v_j)$  is true, the availability of second of these,  $v_j$  relies on or depends on the availability of the first,  $v_i$ . We can represent this model of dependencies as a graph  $G = (V, E)$  where  $V$  is the set of network nodes and  $e = (v_i, v_j) \in E$  means that  $d(v_i, v_j)$  is true. (For example, in the graph below, both H and J depend on F.)


Your team defines a *vulnerable set* to be a subset  $V'$  of the nodes in  $V$  where all nodes in the subset depend on each other either directly or indirectly. (By “indirectly”, we mean that  $d(v_i, v_j)$  is not true, but  $v_j$  depends on another node which eventually depends on  $v_i$ , perhaps through a “chain” of dependencies. For example, in the graph below, F depends on D indirectly.)

**Your problem:** Describe an algorithm (and give its time complexity) that finds the vulnerable set  $VM$  in  $G$  where

1. the number of nodes in  $VM$  is no smaller than the number of nodes in any other vulnerable set found in  $G$ , and
2.  $\nexists v_i \in VM$  and  $v_j \notin VM$  s.t.  $d(v_i, v_j)$

The second condition means that there are no nodes outside of  $VM$  that depend on any node in  $VM$ .

For example, in the graph below, there are a number of vulnerable sets, including  $\{D, E, F\}$ ,  $\{J, K\}$  and  $\{G, H, I\}$ . The first of these does not meet the second condition. The other two do, but the last one is larger than the second one, so  $VM = \{G, H, I\}$ .



vulnerable-sets.png

**Solution:** First, call  $DFS - sweep(G)$  to find finishing times for each node and use that to compute  $G^T$ , the transpose of  $G$ . Next, call  $DFS - sweep(G^T)$  in order of decreasing finishing time according to the times found in  $DFS - sweep(G)$ . For every call to  $DFS - visit()$  in the for loop of  $DFS - sweep(G^T)$ , create a DFS tree for every vulnerable set using the initial node as the root node. This results in a forest of DFS trees that represent the total number of vulnerable sets. Next, to satisfy the second requirement, we must check that there are no dependent node outside of a tree. Look at the root node of a DFS tree and check if it points to a node outside of the tree. If it does, the tree can be eliminated. If it does not, perform the same test on the other nodes. While checking for the second requirement, also keep track of the size of each tree in a list. Repeat this process on all the DFS trees. Lastly, to satisfy the first requirement, return the DFS tree with the largest number of nodes.

We can find all vulnerable sets in  $O(V + E)$  time since we are using DFS. Next, it takes  $O(V)$  time to check every DFS tree for the second requirement since in the worst case every DFS tree is valid, meaning every node must be checked. Lastly, it takes  $O(v \log v)$  to sort the list containing the tree sizes and find the largest tree. Therefore, the total time complexity is  $O(V + E + V + n \log n) = O(2V + E + n \log n)$ .

#### PROBLEM 4 Gradescope Submission

Submit a version of this .tex file to Gradescope with your solutions added, along with the compiled PDF. You should only submit your .pdf and .tex files.