CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, APRIL 8, 2022 AT 11:30 PM VIA GRADESCOPE

BADV(WYZ5RGE)

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: jhs8cue, spj6s

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: Consider a minimum spanning tree of graph G, and assume toward a contradiction that edge e with the smallest weight of all edges in G, is not in the MST.

In the case that all edges in G have unique weights, e not being in the MST of G means that there exists another edge that has a smaller weight than e in the MST, meaning e is not actually the smallest weight edge in G, which is contradictory.

In the case that there exist some edges that have the same weight, then there exist several MST for G. For the same reasoning as above, even if there exist multiple edges that have the same weight as e, there must exist at least one MST of G that includes edge e.

Because assuming that minimum-weight edge e belongs to no MST of G led to a contradiction, it must be the case that e belongs to some minimum spanning tree of G.

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 $hops(a_i, a_k)$ to be the smallest number of flight connections between two airports. They define $maxHops(a_i)$ to be the number of hops to the airport that is farthest from a_i, i.e. $maxHops(a_i) = max(hops(a_i, a_j)) \, \forall a_j \in A$.

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 $maxHops(a_i)$ 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 $maxHops()$ value).

They also define "Core 2 airports" to be the set of airports that have a $maxHops()$ 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: The maxHops method is analagous to the longest path problem. As covered in CS 2150 and 3102, the longest path problem is NP-Complete, and the only known way to go about

solving it is a quadratic runtime brute force approach, $O(n^2)$, or in this case $O(v^2)$ - note that this is the runtime to find the longest path for each airport node. To find every core 1 airport, we have to find the longest path for every airport node using maxHops, and then choose the smallest value. Because we are running our $O(v^2)$ brute force algorithm for every airport node v, the time complexity of getting all maxhHops values is $O(v^3)$. All of these values can be saved in an array and then core 1 airports can be found by searching through the array for the smallest value (or values, if maxHops values are not unique), which is linear at worst. Core 2 airports can then be found by incrementing the smallest maxHops value by 1 and using that value as a search key through the array again. Overall, the runtime is $O((v^2 * v) + v + v) = O(v^3)$.

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 $VM = \{G, H, I\}$.

Solution: This algorithm boils down to identifying strongly connected components in the graph and seeing which SCC has nodes that do not point to nodes outside of the SCC. The identifying and labelling of SCCs is done with the same algorithm described in homework B2basic to identify SCCs and keep track of how many vertices they have, where DFS-Sweep is run on G, then run on

the transpose G^T, starting on the vertex with the largest finish time and continuing in descending order of finish times, with a counter variable initialized to 0, which is assigned to each vertex's *.cc* data field to identify which SCC it belongs to. This counter variable is then incremented every time the algorithm finds a cycle and moves onto the next unvisited vertex, as this newly visited vertex is not party of the previous SCC. In the end, we will be left with a graph of vertices who are assigned an integer value acting as a label for the SCC it belongs to (e.g., a vertex with *.cc* value 0 means it belongs to SCC 0). While this process of identifying SCCs is being carried out, keep a separate array/indexed list data structure; every time our counter variable is assigned to a vertex's *.cc* data field, increment the store value at the corresponding index of our array. This array will then serve as a list of sizes corresponding to that SCC, similar to the indirect heap approach we discussed in class.

After we have identified SCCs and their sizes, we can search the size array for the largest SCC. When the largest SCC is identified, run breadth first search on any of the vertices in it. If BFS returns a path to any vertex outside of the SCC (in other words, any vertex that doesn't have the same *.cc* value), then that means there exists a path from the SCC to an outside vertex, meaning the SCC is not a vulnerable set. If BFS search returns paths from the start vertex to other vertices in the SCC, then that means the SCC is a vulnerable set.

If all SCCs in G are unique in size, then the alogrithm can be stopped here. If there exist multiple SCCs of the same maximum size, then repeat the process of finding them in the sizes array and running BFS on one of the vertices in them.

If none of the SCCs that are the maximum size are determined to be vulnerable sets, then that means there exists no set VM in G, as VM must be both the largest SCC in G, as well as a vulnerable set.

The runtime of this algorithm is $O(EV)$, as BFS search is run on every SCC, which at worst is v number of SCCs

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.