CS4102 - ALGORITHMS - SPRING 2022

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

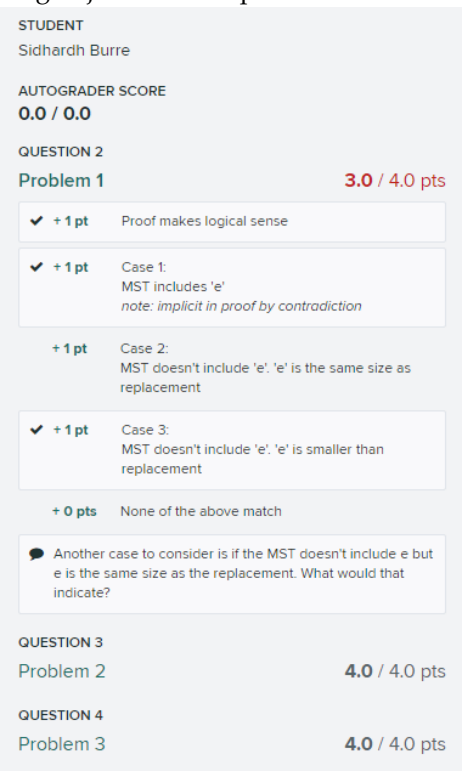
BADV(SSB3VK)

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)*

As according to piazza post with note @351, I will be including an image of the feedback provided for each question. Because I was only marked down for problem 1, I will only be making adjustments to problem 1.



STUDENT
Sidhardh Burre

AUTOGRADER SCORE
0.0 / 0.0

QUESTION 2
Problem 1 — **3.0** / 4.0 pts

- ✔ +1 pt — Proof makes logical sense
- ✔ +1 pt — Case 1: MST includes 'e' *note: implicit in proof by contradiction*
- +1 pt — Case 2: MST doesn't include 'e'. 'e' is the same size as replacement
- ✔ +1 pt — Case 3: MST doesn't include 'e'. 'e' is smaller than replacement
- +0 pts — None of the above match

Another case to consider is if the MST doesn't include e but e is the same size as the replacement. What would that indicate?

QUESTION 3
Problem 2 — **4.0** / 4.0 pts

QUESTION 4
Problem 3 — **4.0** / 4.0 pts

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: *Let us proceed via proof by contradiction.*

We have a incomplete MST (G - v) where we are performing the final addition to complete the MST. We have two methods of adding v, via e which goes from u to v and is the minimum-weight edge (the edge with the smallest weight) and via e' which goes from u' to v. Because e is the minimum weight edge, w(e) $<$ w(e'). If we were to add the edge e' to the MST of G - v, then the

cumulative weights would be w(G - v) + w(e'). This value is not the MST because we stated that e is the minimum weight edge and therefore there exists a MST with weight < w(G - v) + w(e'). Specifically, this weight is w(G - v) + w(e) because w(e) < w(e'). Therefore, the MST must include edge e instead of any other edge (e' in this case). In the case where w(e) = w(e') the selection between e and e' is irrelevant, either edge would produce the MST.

To be more specific, in the case that the MST doesn't include the edge e but e is the same size as the alternative edge that the algorithm chose as the replacement, then we could use the exchange argument to state that the other alternate choice (with equal edge-weight) would not leave us any worse of simply for not selecting the edge e.

To rephrase this argument in my terms, consider the case where w(e) = w(e') and both e and e' span the gap between MST(G - v) and v. In this case, our edge e doesn't necessarily belong in the MST. The minimum weight edge definition is not singular, it can also apply to multiple edges (in this case e'). Therefore, the selection between e and e' is arbitrary and is irrelevant to the integrity of the MST.

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: This problem can be reduced to finding the set of Core 1 airports. This is because the set of Core 2 airports are simply the children of the Core 1 airports (excluding the Core 1 airports themselves). Therefore the real objective for this problem is to find the nodes that are the most evenly far away from the other nodes as possible.

A brute force method of computing this problem would be to compute BFS/DFS on each node and then store the maximum distance for each node to any other node (the maxHops value). But that seems inefficient and I am almost a 100% sure there is a more efficient way to go about it.

A couple cases that expose weaknesses in any algorithm for this problem:

1. the case where all nodes are arranged in a circle
2. the case where all nodes are arranged in a line
3. the case where nodes are arranged in a spider-web fashion
4. the case where nodes are in a bike-spoke fashion

Any algorithm selected must at least be able to provide the right answer in the above cases. The problem is simple when nodes are arranged in a line (either cyclically or linearly) where there is only one possible path to pursue.

I have a solution to this problem. We perform two BFS's. The first BFS is used to identify a "cliff" node. This is a node that is at the edge of the graph. We then perform another BFS on this cliff node. And this time we record a path from the second call to BFS to the final element that was blacked. Then we identify the node that is at the center of the path to be the Core 1 node.

Note that this has a major problem when we have strongly connected sets of nodes that are greater than 3 members in size. Specifically, consider the case that we have a trio of points that make an SCC, in the center of those three points, there is another point that can connects to each point in the trio. If each of the three trio nodes has edges toward two other nodes, then the middle point will be identified as the Core 1 point even though the trio nodes are also Core 1 points.

A potential solution to this is to identify SCCs and then collapse those nodes but, in the case that we have overlapping SCCs, this process becomes a lot harder.

I'm going to be a coward and just use the trivial method. This method simply calls BFS on every single vertex, recording the longest optimized distance to every node. Then, I will select the shortest longest optimized distance vertexes from these vertex, longest optimized distance pairs. These vertices will composed the Core 1 nodes. All nodes with value 1 greater than the value for the Core 1 nodes will be Core 2 nodes.

We ultimately perform V BFS searches and then iterate over the list, selecting the Core 1 and Core 2 nodes. This leads to an overall $\Theta(V(V+E)+V)$ run time. If we simplify this down, we get $\Theta(V^2 + V * E)$.

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.)

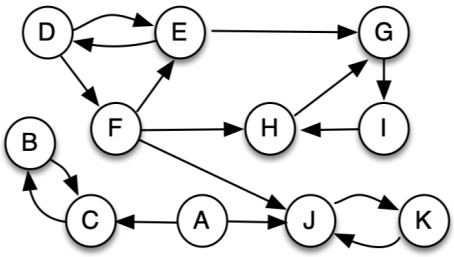
Your team defines a *vulnerable set* to be a subset V' of the nodes in V where all nodes in the subset depend on each other either directly or indirectly. (By "indirectly", we mean that $d(v_i, v_j)$ is not true, but v_j depends on another node which eventually depends on v_i, perhaps through a "chain" 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: All this question asks is to find the largest Strongly Connected Component. So we can use a modified version of DFS to output the set VM in G. This modified version performs most of the algorithm normally, but on the second DFS call, the one that is performed on DFS G^T, we keep track of the number of nodes in the Strongly Connected Component, and updating the set containing the largest Strongly Connected Component as we go.

A short description of the second DFS call is demonstrated below:

```
DFS-sweep(G)
    for each vertex u in toposort-list
        u.color = white
        u.pi = NIL
    time = 0
    setVM = {}
    for each vertex u in toposort-list
        if u.color == WHITE
        [setVMCheck] = DFS-VISIT(G^T, u, {u})

        for each vertex u in setVMCheck
            if G.adj[u] has members not in setVMCheck
                continue

        if setVM.size() < setVMCheck.size():
            setVM = setVMCheck
    return setVM


DFS-VISIT(G, u, setVM)
    time = time + 1
    u.d = time
    u.color = GRAY
    for each v in G.Adj[u]
        if v.color == WHITE
            v.pi = u
            setVM.add(v)
            DFS-VISIT(G, v, setVM)
    u.color = BLACK
    time = time+1
    u.f = time
    return setVM
```

Solution: The only added operations to DFS_Sweep we have are the initialization of the setVM object, checking if any component of the SCC has an out-degree, and a comparison (and potentially an assignment). In DFS-Visit, we only add an addition to a set.

Because none of these operations have a non-linear affect on the algorithm (have an operation that is performed $o(n)$), we can ensure that there is no increase in the asymptotic runtime of the algorithm. Note that the "for each vertex u in setVMCheck" only runs once per node so we don't have any increase in our runtime.

Note an important thing that we're doing here. Prior to this function, we have a call of DFS on normal G that gets us our finish times. This second call, is run on G^T using the reverse ordering

of finish times from the first call to select nodes we do DFS-VISIT on. But, the second for loop, uses the adjacency list of G (NOT G^T) to ensure that our solution is a valid SCC that does not have an out-degree.

In terms of runtime, the real change to runtime originates from the check to ensure that the component of the SCC does not connect to a vertex outside the SCC. This check is run only once per node (thankfully) but it can run V times (in the case that the entire graph is a SCC and everything is connected) because we would have to iterate over each node adjacent to the node that we are on and ensure that each of those nodes are in the SCC (could be accomplished with a method similar to the u.cc problem in basics). Or to put it another way, the check can run E times, once for each edge because from each node we might have to check each of its connections and ensure that all of its connections are in the SCC. Note that this increase in runtime is equivalent to the runtime contribution from each call to DFS-VISIT (E). So the overall runtime would be $\Theta(V + E + E)$ which is the same as the original runtime, $\Theta(V + E)$

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.