CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, MARCH 18, 2022 AT 11:30 PM VIA GRADESCOPE

B1 BASIC (SOLUTIONS)

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 *Short Questions on BFS*

A. What is the maximum number of vertices that can be on the queue at one time in a BFS search? Briefly explain the situation that would cause this number to be on the queue.

Solution: $n - 1$ vertices can be on the queue, if the starting vertex has an edge to every other vertex.

B. If you draw the BFS tree for an undirected graph G, some of the edges in G will not be part of the tree. Explain why it's not possible for one of these non-tree edges to connect two vertices that have a difference of depth that's greater than 1 in the tree.

Solution: If a non-tree edge connects v to vertex w then it's possible they're at the same depth (when there are paths of the same length from the start vertex to both vertices). But if v is higher than w in the BFS tree, then the difference in their depths can be at most 1, since otherwise BFS would have chosen the non-tree edge (v, w) as a tree edge when it was processing the vertices adjacent to v.
(This would not be true if the graph was directed.)

PROBLEM 2 *True or False. (You don't have to explain this in your submission, but you should understand the reason behind your answer.)*

A. If you use BFS to detect a cycle in an undirected graph, an edge that connects to a vertex that's currently on the queue or has been removed from the queue indicates a cycle as long as that vertex is not the parent of the current node.

Solution: True. Both types of vertices have been seen or discovered, and thus edges to them are non-tree edges which indicate cycles in an undirected graph.

B. If you use DFS-visit on a *directed* graph with $V > 1$ starting at vertex v_1, you will always visit the same number of vertices that you would if you started at another node v_2.

Solution: False. The direction of edges affect this. Consider a graph with two vertices and one directed edge between them. In a directed graph, vertex v may be reachable from w but not *vice versa*.

C. If you use DFS-visit on a connected *undirected* graph with $V > 1$ starting at vertex v_1, you will always visit the same number of vertices that you would if you started at another node v_2. (If it were not connected, would you answer change?)

Solution: True. In a connected undirected graph, if vertex v is reachable from w, then w is reachable from v. If the graph were not connected, it could be false because v could be in a different connected component than w.

PROBLEM 3 *Finding Cycles Using DFS*

In a few sentences, explain how to recognize a directed graph has a cycle in the DFS-visit algorithm's code we saw in class. How does this need to be modified if the graph is undirected?

Solution: Referring to CLRS's code for DFS-visit, when visiting vertex u there is a for-loop that looks at each of u's neighbors. If one of those vertices v has color gray, then the edge (u, v) is a back-edge and thus part of a cycle. If we add an else-if clause to the if-statement in that loop, we can do whatever we want to do when we recognize a cycle (set a flag, print the edge, at it to a list, etc.)
If the graph is undirected, then we must also make sure that v is not u's parent. So the else-if clause would start:
`else if v.color == GRAY and v != u.pi`
where $u.pi$ is the parent of u in the DFS tree.

PROBLEM 4 *Finding a path between two vertices*

Describe the modifications you would make to DFS-visit() given in class to allow it to find a path from a start node s to a target node t. The function should stop the search when it finds the target and return the path from s to t.

Solution:

1. Add an additional parameter, tgt, to DFS-visit.
2. Modify DFS-visit() so that it returns something to indicate the search has found the target node. There are options, but here we'll return True if the target is found and False otherwise.
3. At the start of DFS-visit() check to see if you've found the target:
   `if (u == tgt) return True`
4. If the recursive call later in DFS-visit() succeeds, then return True immediately.
   `if DFS-visit(G, v, tgt) return True` Otherwise continuing searching other neighbors of u.
5. If you get to the last line in the function, you didn't find tgt, so return False.
6. To use this to find the path, call DFS-visit() and pass it s as the start node and t as the new parameter tgt. If it returns True, you can loop starting at t and use each node's pi value to trace from parent to parent until you get to s — this give you all the vertices in the path (visited in reverse order).

PROBLEM 5 *Labeling Nodes in a Connected Components*

In a few sentences, explain how you'd modify the DFS functions taught in class to assign a value $v.cc$ to each vertex v in an undirected graph G so that all vertices in the same connected component have the same cc values. Also, count the number of connected components in G.
In addition to your explanation, give the order-class of the time-complexity of your algorithm.

Solution:

1. Add an additional parameter, cc_num, to DFS-visit(). In DFS-visit(), assign this value to $u.cc$.
2. Modify DFS-sweep() to have a variable cc_ctr that's initialized to 0.
3. Before calling DFS-visit(), increment cc_ctr and pass that as the new parameter.
4. When the loop in DFS-sweep() finishes calling DFS-visit() on all unvisited vertices, all vertices will be numbered and the variable cc_ctr will be the count of the connected components in G.
5. This does not change the cost of DFS-sweep, so it's $\Theta(V + E)$.

PROBLEM 6 *BFS and DFS Trees*

Consider the BFS tree T_B and the DFS tree T_D for the same graph G and same starting vertex s. In a few sentences, clearly explain why for every vertex v in G, the depth of v in the BFS tree cannot be greater than its depth in the DFS tree. That is:

$$\forall\, v \in G.V, depth(T_B, v) \leq depth(T_D, v)$$

(Here the depth of a node is the number of edges from the node to the tree's root node. Also, you can use properties of BFS and DFS that you've been taught in class. We're not asking you to prove those properties.)

Solution: Let's assume the opposite of that is true: $\exists\, v \in G.V, depth(T_B, v) > depth(T_D, v)$
This means there is a path in the DFS tree T_D from s to v that has fewer edges than the path from s to v in the BFS tree T_B. But this is not possible, because we know that BFS finds the shortest possible path from a start node to any other node. (Don't forget the paths in both trees are made up of edges from G so paths in the trees are in G too.) Thus we've proved the original claim (using proof by contradiction).

PROBLEM 7 *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.