DSC 40B - Homework 06

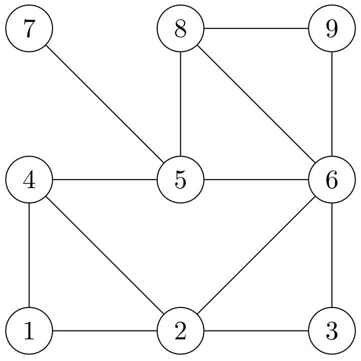
Due: Wednesday, February 25

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope at 11:59 p.m.

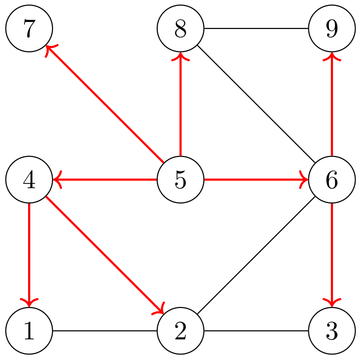
Problem 1.

For the following problems, recall that (u, v) is a *tree edge* if node v is discovered while visiting node u during a breadth-first or depth-first search. Assume the convention that a node's neighbors are produced in ascending order by label. You do not need to show your work for this problem.

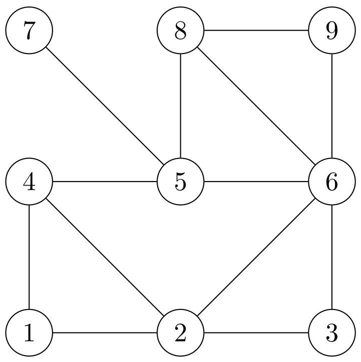
a) Suppose a breadth-first search is performed on the graph below, starting at node 5. Mark every BFS tree edge with a bold arrow emanating from the predecessor.



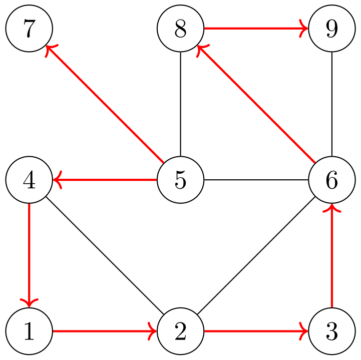
Solution:



b) Suppose a depth-first search is performed on the graph below, starting at node 5. Mark every DFS tree edge with a bold arrow emanating from the predecessor.



Solution:



c) Fill in the table below so that it contains the start and finish times of each node after a DFS is performed on the above graph using node 5 as the source. Begin your start times with 1.

Node	Start	Finish
1		
2		
3		
4		
5		
6		
7		
8		
9		

Solution:

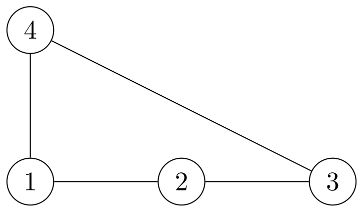
Node	Start	Finish
1	3	14
2	4	13
3	5	12
4	2	15
5	1	18
6	6	11
7	16	17
8	7	10
9	8	9

Problem 2.

Suppose s, u and v are three distinct nodes in a **directed unweighted** graph, and that v is a neighbor of u (i.e, there is a direct edge (u, v) from u to v).

a) (True or False): Suppose you run the BFS algorithm as described in class from source node s. Suppose at some point during the algorithm, the status of u is 'undiscovered'. Then the status of v must be 'undiscovered'. [You don't need to justify your answer.]

Solution: False, consider the following graph:



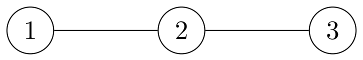
Let $s = 1, u = 3, v = 4$. After one pass, v will be discovered, but u will still be undiscovered.

b) (Fill in the blanks) Suppose the shortest path distance from s to u is 10. Then the smallest possible shortest path distance from s to v is ______________

Solution: 1. There can be an edge from s to v, and an edge from u to v, but not an edge from v to u.

c) (True or False): Suppose you run the DFS algorithm as described in class from source node s. Suppose at some point during the algorithm, the status of v is 'pending'. Then it is possible that the status of u is 'visited'. [You don't need to justify your answer.]

Solution: True, consider the following graph:



Let $s = 1, u = 3, v = 2$. After two passes, v will be pending while u is visited.

Problem 3.

Consider the following code (which does not do anything meaningful) that take **an undirected and connected** graph $G = (V, E)$ as input. Assume that the graph is represented by the adjacency list representation.

Here 'dfs' within the code is the same DFS procedure that we described in class (e.g, on slide 6 of Lecture 13).

Analyze its time complexity in terms of $|V|$ and $|E|$ using asymptotic language. Make your answer as tight as possible, and loose bound receives fewer points. [You don't need to justify your answer.]

```python
def SomeGraphFunction(G):
    for u in G.nodes:
        for v in G.neighbors(u):
            status = {node: 'undiscovered' for node in G.nodes}
            dfs(G, v, status)
            print("Hello!")
```

Solution: $\Theta(|E||V| + |E|^2)$.

DFS has time complexity $\Theta(|V| + |E|)$. The graph is undirected and connected, therefore this function loops through all neighbors $2|E|$ times (if every neighbor of every node is visited, then each edge is visited twice). Thus the total time complexity is $\Theta(|E|(|V| + |E|)) = \Theta(|E||V| + |E|^2)$.

Problem 4.

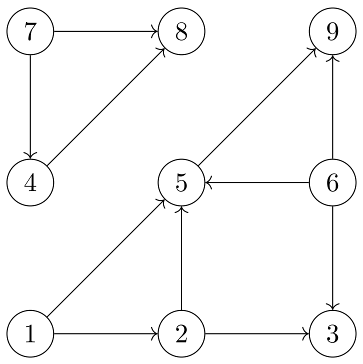
Give an example of a DAG $G = (V, E)$ of $|V| = n$ nodes, yet with $|E| = \Theta(n^2)$ edges.

Solution: We can construct such a DAG like this: let the first node have neighbors $2, \ldots n$, the second node have neighbors $3, \ldots, n$, etc. such that the ith node has neighbors $i + 1, \ldots, n$ (the last node has no outgoing edges). Thus, this is a DAG with $1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$ edges, which is $\Theta(n^2)$.

Problem 5.

Topologically sort the vertices of the following graph. Note that there may be multiple, equally-correct topological sorts.

You do not need to show your work for this problem.



Solution: Our algorithm for producing a topological sort of the nodes is to first perform a DFS in order to record finish times, and then to sort the nodes in order of decreasing finish time.

If we start a DFS at node 1 and us the convention that the neighbors are produced in increasing order of label, we will visit nodes 1, 2, 3, 5, and 9. This search didn't visit all of the nodes of the graph, so suppose we restart the search at node 6. This search will not move away from 6, and so we restart the search again at node 7, visiting nodes 4 and 8. The finish times resulting from this search are

```
times.finish = {
    1: 10,
    2: 9,
    3: 4,
    4: 17,
    5: 8,
    6: 12,
    7: 18,
    8: 16,
    9: 7
}
```

Ordering these by decreasing finish time, we obtain a topological sort of:

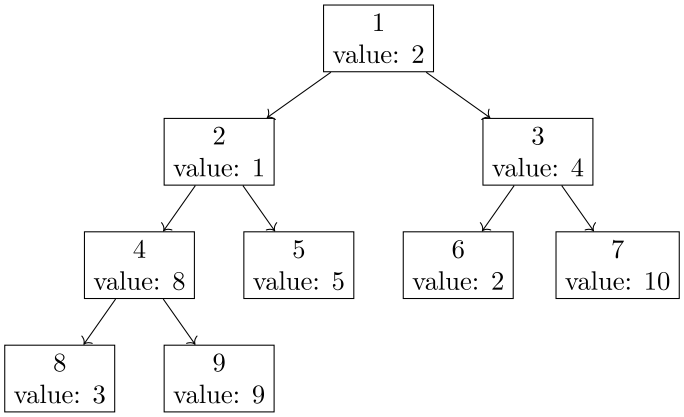
$$7, 4, 8, 6, 1, 2, 5, 9, 3.$$

Your topological sort may be different, but still correct. This may happen if you restarted the search at different nodes than what was used above.

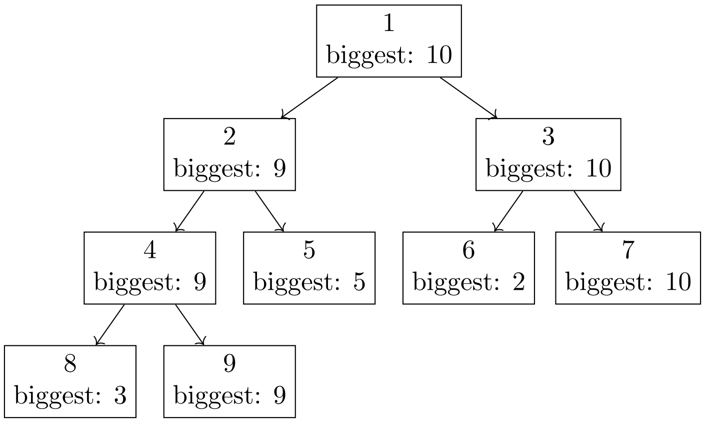
Programming Problem 1.

You are given a directed graph representing a tree and a dictionary `value` which contains a value for each node. Define the *biggest descendent value* of a node u to be the largest value of any node which is a descendent of u in the tree (for this problem, you should consider u to be a descendent of itself.

For instance, given the following tree where each node's label is replaced by its value:



The *biggest descendent value* for each node is:



In a file named `biggest_descendent.py`, write a function `biggest_descendent(graph, root, value)` which accepts the graph, the label of the root node, and the dictionary of values and returns a dictionary mapping each node in the graph to its biggest descendent value.

The input graph will be an instance of `dsc40graph.DirectedGraph()`.

Example:

```
>>> edges = [(1, 2), (1, 3), (2, 4), (2, 5), (4, 8), (4, 9), (3, 6), (3, 7)]
>>> g = dsc40graph.DirectedGraph()
>>> for edge in edges: g.add_edge(*edge)
>>> value = {1: 2, 2: 1, 3: 4, 4: 8, 5: 5, 6: 2, 7: 10, 8:3, 9: 9}
>>> biggest_descendent(g, 1, value)
{1: 10, 2: 9, 3: 10:, 4: 9, 5: 5, 6: 2, 7: 10, 8: 3, 9: 9}
```

Solution: We use DFS. We don't need to check whether a neighbor has been discovered when searching a tree, because there are no cycles.

```
import dsc40graph

def biggest_descendent(graph, root, value, biggest=None):
    if biggest is None:
        biggest = {}

    biggest[root] = value[root]

    for v in graph.neighbors(root):
        biggest_descendent(graph, v, value, biggest)
        if biggest[v] > biggest[root]:
            biggest[root] = biggest[v]

    return biggest
```