DSC 40B - Homework 07

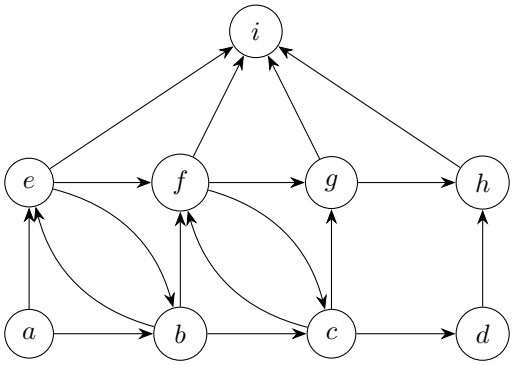
Due: Wednesday, November 19

Write your solutions to the following problems by 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.

In lecture, we saw that breadth-first search (BFS) can be used to find the shortest paths from a source node to all other nodes in an unweighted graph. Our implementation of BFS returned two dictionaries: `distance`, containing the shortest path distance from the source node to each node in the graph, and `predecessor`, containing the BFS predecessor of each node.

Consider a breadth-first search on the graph shown below, starting with node a.



Write down the `distance` and `predecessor` dictionaries returned by BFS on this graph.

Note: you should adopt the convention that `.neighbors()` returns the neighbors of a node in **ascending alphabetical order**.

Solution:

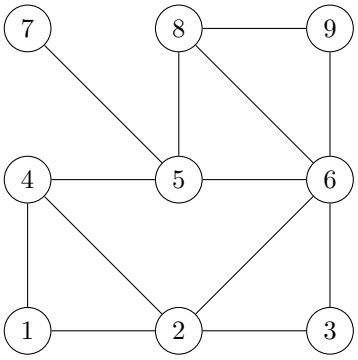
```
distance = {
    'a': 0,
    'b': 1,
    'c': 2,
    'd': 3,
    'e': 1,
    'f': 2,
    'g': 3,
    'h': 4,
    'i': 2
}

predecessor = {
    'a': None,
    'b': 'a',
    'c': 'b',
    'd': 'c',
    'e': 'a',
```

```
    'f': 'b',
    'g': 'c',
    'h': 'd',
    'i': 'e'
}
```

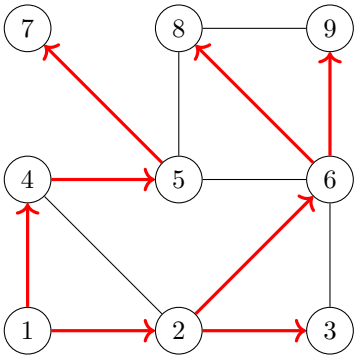
Problem 2.

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.

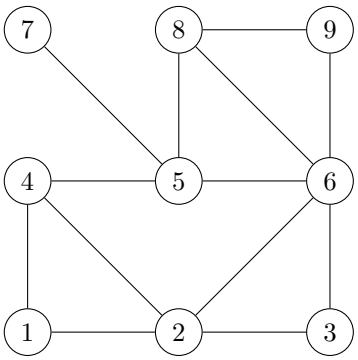
a) Suppose a breadth-first search is performed on the graph below, starting at node 1. Mark every BFS tree edge with a bold arrow emanating from the predecessor.



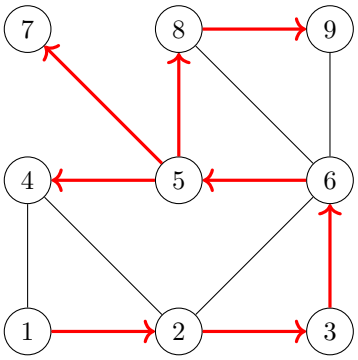
Solution:



b) Suppose a depth-first search is performed on the graph below, starting at node 1. 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 1 as the source. Begin your start times with 1.

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

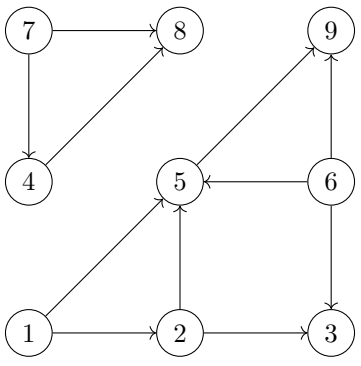
Solution:

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

Problem 3.

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:

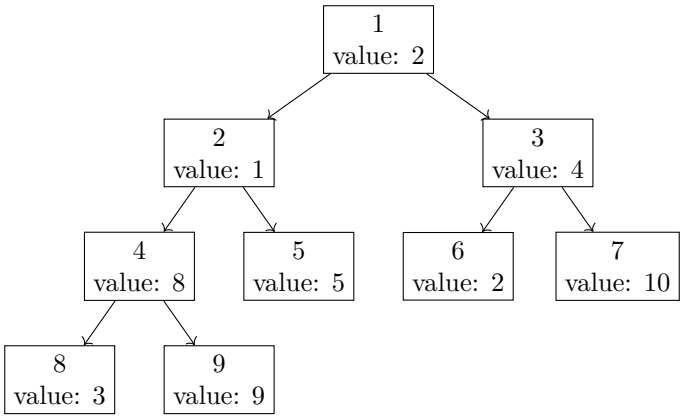
$$7, 4, 8, 6, 1, 2, 5, 9, 3.$$

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.

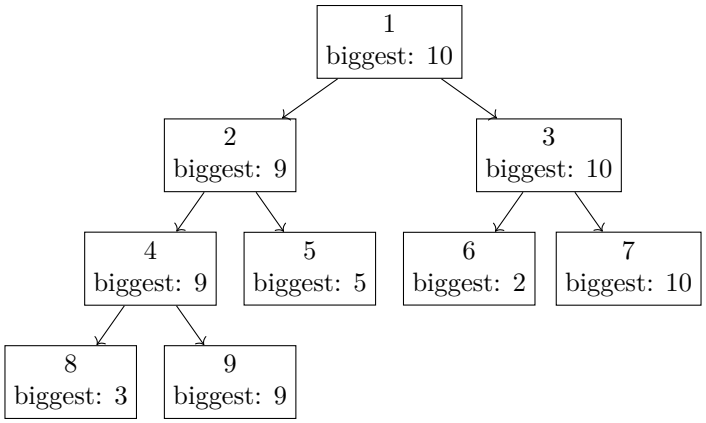
Programming Problem 1.

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
```

Programming Problem 2.

Suppose we are given a weighted, undirected graph $G = (V, E, \omega)$ in which the edge weights represent *similarity*; for example, the similarity between two users in a social network. Given a number λ, which we call the *level*, we will say that the *clusters* of G are the connected components of the graph after all edges whose weight is less than λ have been removed. There are other ways of defining the clusters of a weighted graph, but this is one natural way.

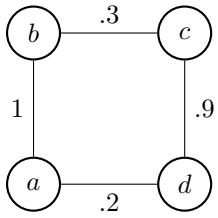
In `cluster.py`, write a function `cluster(graph, weights, level)` which computes the clusters of a

weighted graph. Here, `graph` is an instance of `dsc40graph.UndirectedGraph`, `weights(u, v)` is a function returning the weight of edge (u, v), and `level` is a number representing the level at which to find the clusters. Its return value should be a `frozenset`[1] containing `frozensets`; the inner `frozensets` should contain the nodes in a cluster.

Your code should not modify the graph in any way and it should not create a copy of the graph. It should run in $\Theta(V + E)$ time.

Note that `weights` is a function that will be constructed by us and passed to your function; you will not need to create it (except in your own tests).

For example, given the following graph:



The output when run with a level of 0.4 should be:

```
>>> def weights(x, y):
... x, y = (x, y) if x < y else (y, x)
... return {("a", "b"): 1, ("b", "c"): .3, ("c", "d"): .9, ("a", "d"): .2}[(x, y)]
>>> cluster.cluster(graph, weights, 0.4)
frozenset([frozenset(['a', 'b']), frozenset(['c', 'd'])])
```

Note: you might see curly braces instead of square brackets in the output – that's OK. Different versions of Python print frozensets differently.

Solution: For this problem, we can use a modified BFS or DFS (solution is BFS) to find the connected components or clusters while ignoring edges that are below the weight threshold. Returning elements visited is important as scanning the status array for changes can lead to a higher runtime of $\Theta(V^2 + E)$ This approach takes $\Theta(V + E)$ time in total.

```
def cluster(graph, weights, level):
    status = {node: 'undiscovered' for node in graph.nodes}
    #list to store frozensets
    lst = []
    for node in graph.nodes:
        if status[node] == 'undiscovered':
            # These nodes were visited on one iteration of BFS
            # meaning they are connected
            visited = bfs(graph, node, weights, level, status)
            # append these connected elements
            lst.append(frozenset(visited))
    return frozenset(lst)

def bfs(graph, source, weights, level, status=None):
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}
    status[source] = 'pending'
    pending = deque([source])
```

[1] We use `frozensets` here because clusters are sets, but `sets` cannot be elements within other `sets` since they are not hashable.

```
    # create visited array to hold changed elements
    visited = []
    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v) only if weight is greater than level
            if weights(u,v) < level:
                continue

            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
        visited.append(u)
    return visited
```