DSC 40B - Homework 07

Due: Wednesday, February 24

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 on Wednesday at 11:59 p.m.

Wherever necessary in this homework, you should adopt the convention that a node's neighbors are produced in ascending order of label.

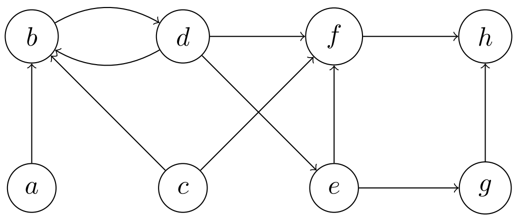


Figure 1: Graph for Problem 1.

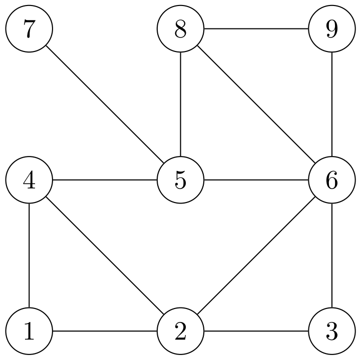
Problem 1. *(Eligible for Redemption)*

Consider a *breadth*-first search on the graph shown in Figure 1, starting with node c. For each node in the graph, write down the distance and predecessor found by the BFS.

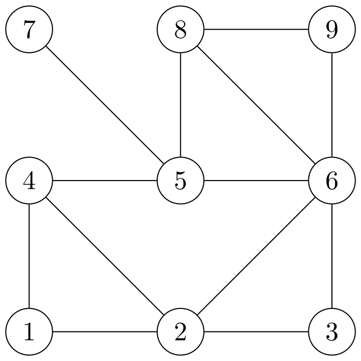
Problem 2. *(Eligible for Redemption)*

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.



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.

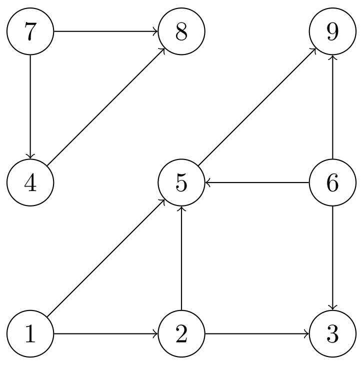


c) Fill in the table below so that it contains the start and finish times of each node after a DFS is performed on the above graph using node 1 as the source. Begin your start times with 1.

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

Problem 3.

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

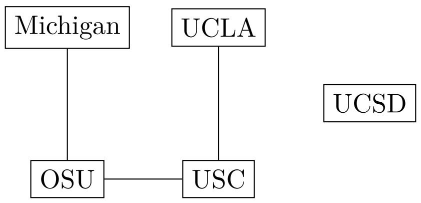


The following programming problems will ask you to write functions which accept a graph as input. The graph will be an instance of the `UndirectedGraph` class (or `DirectedGraph` class, if the problem says so) from the `dsc40graph` module. You can install this module on your own computer by running `pip install dsc40graph`. Alternatively, you can download the module from https://raw.githubusercontent.com/eldridgejm/dsc40graph/master/dsc40graph.py and place it in the same directory as your code. Or, if you'd prefer, you can simply use DataHub to write your code – we have pre-installed `dsc40graph` for you.

The documentation for the graph library, as well as examples of it in action, can be found at https://eldridgejm.github.io/dsc40graph/. This graph class behaves exactly like the graphs we have seen in lecture; namely it has a `.neighbors()` method and a `.nodes` attribute.

Programming Problem 1.

We can use a graph to represent rivalries between universities. Each node in the graph is a university, and an edge exists between two nodes if those two schools are rivals. For instance, the graph below represents the fact that OSU and Michigan are rivals, OSU and USC are rivals, UCLA and USC are rivals, but UCSD does not have a rival.



In a file called `assign_good_and_evil.py`, write a function `assign_good_and_evil(graph)` which determines if it is possible to label each university as either "good" or "evil" such that every rivalry is between a "good" school and an "evil" school. The input to the function will be a graph of type `UndirectedGraph` from the `dsc40graph` package. If such a labeling of "good" and "evil" exists, your function should return it as a dictionary mapping each node to a string, `'good'` or `'evil'`; if a labeling doesn't exist, it should return `None`. For example, such a labeling does exist for the above graph:

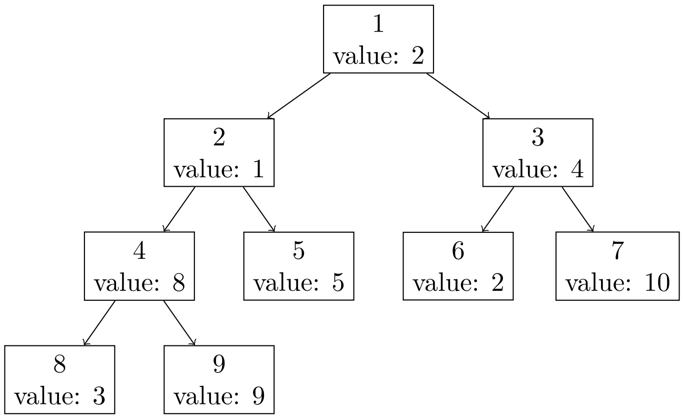
```
>>> example_graph = dsc40graph.UndirectedGraph()
>>> example_graph.add_edge('Michigan', 'OSU')
>>> example_graph.add_edge('USC', 'OSU')
>>> example_graph.add_edge('USC', 'UCLA')
>>> example_graph.add_node('UCSD')
>>> assign_good_and_evil(example_graph)
{
    'OSU': 'good',
    'Michigan': 'evil',
    'USC': 'evil',
    'UCLA': 'good',
    'UCSD': 'good'
}
```

Your algorithm should work for a general graph – not just the example shown above. The input graph will be of type `UndirectedGraph` from `dsc40graph.py`.

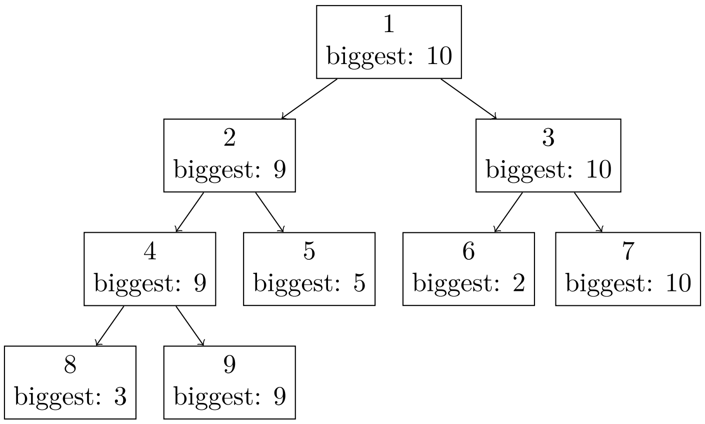
Programming Problem 2.

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 computes a dictionary mapping each node in the graph to its biggest descendent value.

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