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.

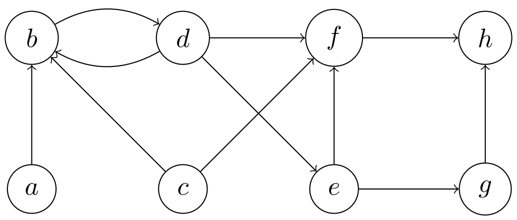


Figure 1: Graph for Problem 1.

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.

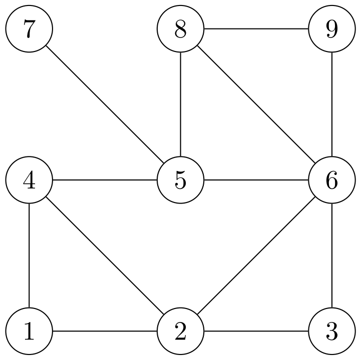
Solution:

```
>>> distance
{
    'a': float('inf'),
    'b': 1,
    'c': 0,
    'd', 2,
    'e', 3,
    'f', 1,
    'g', 4,
    'h', 2
}
>>> predecessor
{
    'a': None,
    'b': 'c',
    'c': None,
    'd': 'b',
    'e': 'd',
    'f': 'c',
    'g': 'e',
    'h': 'f'
}
```

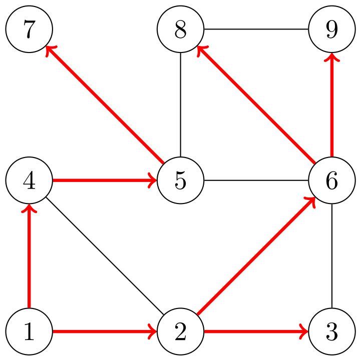
Problem 2. *(Eligible for Redemption)*

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.

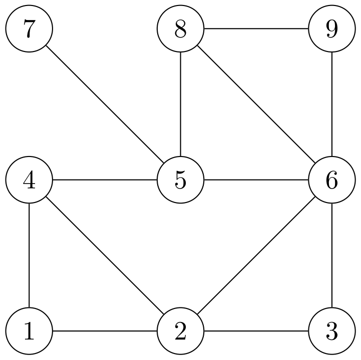
a) Suppose a breadth-first search is performed on the graph below, starting at node 1. Mark every BFS tree edge with a bold arrow emanating from the predecessor.



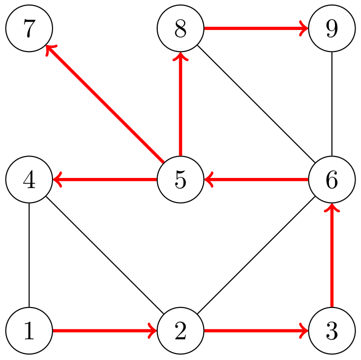
Solution:



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.

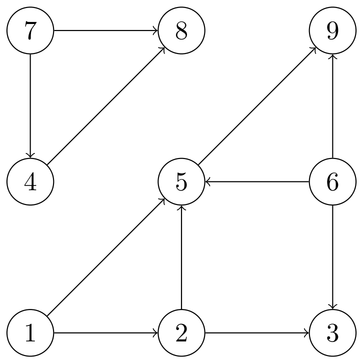
Node	Start	Finish
1		
2		
3		
4		
5		
6		
7		
8		
9		

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.



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:

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

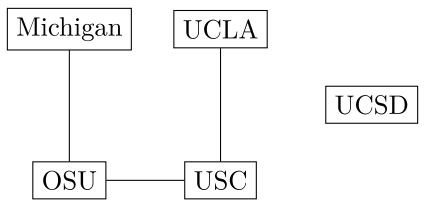
Your topological sort may be different, but still correct. This may happen if you restarted the search at different nodes than what was used above.

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 there is a way to label each node as "good" and "evil" so that every rivaly is between a "good" school and an "evil" school, your function should return it as a dictionary mapping each node to a string, `'good'` or `'evil'`; if such a labeling is not possible, your function should return `None`.

For example:

```
>>> 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'
}
```

Of course, there might be several ways to label the graph – your code need only return one labeling.

Solution: This can be solved using a graph search algorithm. We start by labeling the source node as either "good" or "evil" – it doesn't matter which. We then label all of its neighbors with the opposite label. From there on, any time we visit a node and loop through the neighbors, we give each neighbor a label that is opposite the label of the node we are currently visiting.

When looking through the neighbors, however, we may come across a node which already has already been assigned a label. If that label is the opposite of label of the node we are currently visiting, then all is fine. If the two labels are the same, then we have discovered that it is not possible to label the graph in this way.

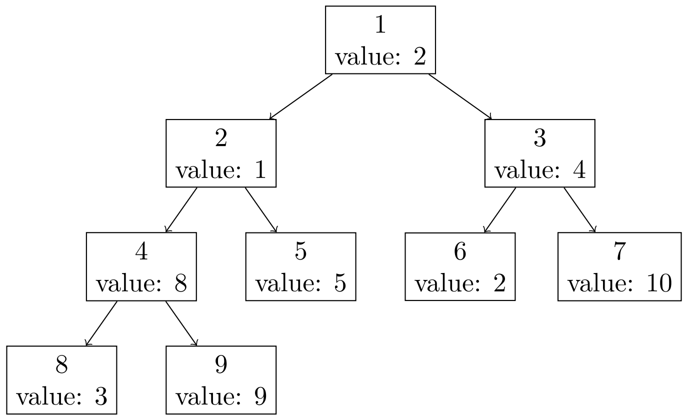
We will implement this idea by modifying the "full" BFS. We use the "full" version of the algorithm because the graph may not necessarily be connected (the example graph is not), yet we still wish to assign labels to every node. The code is shown below:

```python
from collections import deque

def assign_good_and_evil(graph):
    label = {}
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered':
            possible = good_and_evil_bfs(graph, node, status, label)
            if not possible:
                return None

    return label

def good_and_evil_bfs(graph, source, status, label):
    status[source] = 'pending'
    label[source] = 'good'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                if label[u] == 'good':
                    label[v] = 'evil'
                else:
                    label[v] = 'good'
                # append to right
                pending.append(v)
            elif label[u] == label[v]:
                return False
        status[u] = 'visited'

    return True
```

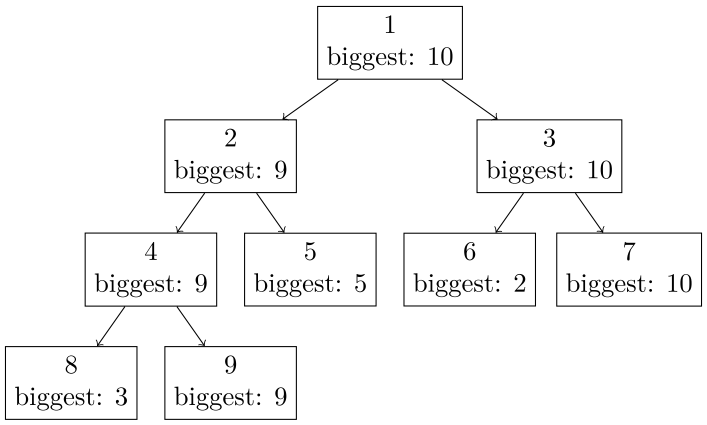
Programming Problem 2.

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

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

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
import pprint


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


if __name__ == '__main__':
    graph = dsc40graph.DirectedGraph()

    edges = [(1,2), (2,4), (2,5), (4,8), (4,9), (1,3), (3,6), (3,7)]
    for edge in edges:
        graph.add_edge(*edge)

    values = {
        1: 2,
        2: 1,
        3: 4,
        4: 8,
        5: 5,
        6: 2,
        7: 10,
        8: 3,
        9: 9
    }

    biggest = biggest_descendent(graph, 1, values)
    pprint.pprint(biggest)
```