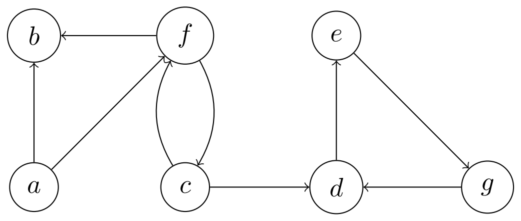
DSC 40B - Discussion 06

Problem 1.



a) Consider a *breadth*-first search on the graph shown in the figure, starting with node c. Which nodes are visited, and in what order? Use the convention that graph.neighbors() produces successors in ascending order of label.

b) Consider a *breadth*-first search on the graph shown in the figure, starting with node a. Which nodes are visited, and in what order? Use the convention that graph.neighbors() produces successors in ascending order of label.

c) Consider a *breadth*-first search on the graph shown in the figure, starting with node g. Which nodes are visited, and in what order? Use the convention that graph.neighbors() produces successors in ascending order of label.

Problem 2.

Consider line 6 in Algorithm 1 shown below in Figure 1 Whenever this line is executed, is it possible...

a) ...that node is 'pending'?

b) ...that node is 'visited'?

c) ...that some other node is 'pending'?

d) ...that some other node is 'visited'?

Problem 3.

It turns out that BFS does not need to use three distinct statuses (undiscovered, pending, visited) in order to function – we just use three to help us understand the algorithm. In fact, we can delete code from bfs, and the function will still work. List below the line number(s) which can all be deleted without affecting the behavior of the algorithm.

Problem 4.

In a unweighted graph, does BFS (with minimal modifications) guarantee that you'll find a shortest path from the starting node u to to any other node v that is reachable from u?

Problem 5.

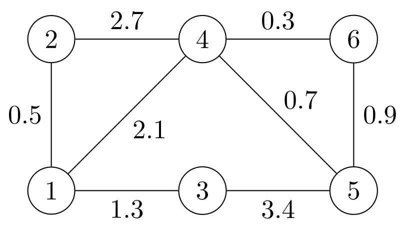
An *edge weighted graph* $G = (V, E, \omega)$ is a graph along with a function $\omega : E \to \mathbb{R}$ which assigns a weight to every edge in the graph. One of the uses of edges weights is to encode dissimilarities. That is, the greater the weight of an edge, the more dissimilar the nodes at either end.

Figure 1: Algorithm 1

```python
from collections import deque

def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)

def bfs(graph, source, status=None):
    """Start a BFS at `source`."""
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    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'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

A natural task involving weighted graphs is to *cluster* the nodes of the graph into groups such that nodes in the same group are similar to one another while two nodes in different groups are dissimilar.

Here is a simple way of clustering a weighted, undirected graph $G = (V, E, \omega)$. Given a real number t, place two nodes u and v in the same cluster if (and only if) there is a path between u and v along which every edge has weight $\leq t$. For instance, consider the graph below:



If $t = 1$, there are three clusters: $\{1, 2\}$, $\{4, 5, 6\}$ and $\{3\}$. If $t = 2$, there are two clusters: $\{1, 2, 3\}$ and $\{4, 5, 6\}$. And if $t = 3$, there is one cluster containing all of the nodes.

Design an algorithm which returns the clusters given an input graph, a weight function, weights, and a threshold t. Your algorithm should take $\Theta(V + E)$ time. To receive full credit, your algorithm should not modify the graph or create a copy of it. Provide pseudocode (or Python).