Homework #7

Megan Ku
Data Structures and Algorithms

March 29, 2020

1. Topological Ordering

One way to find a topological ordering in a directed graph is to start at any node and traverse any possible edges, in a depth-first search manner, until a node is reached with no adjacent nodes. The nodes can then be popped on a stack in reverse order. We can continue this process with all other unvisited nodes, popping traversed nodes onto the stack if they are not already in the stack. The data structures needed for implementation include a hash map to keep track of visited/unvisited nodes, a stack to execute depth-first search that resets for each new starting node, and a stack to store the final order that gets built up as we visit each node. Let's walk through an example. In Figure 1, assume the first node we visit is 1. Using depth first search (DFS), we will build the stack (from bottom to top) 1-3-4-5. Since none of those have been visited before, we can mark them all as visited and pop them from our DFS stack and push them to our final stack. So right now, the final stack, from bottom to top, looks like 5-4-3-1. The only unvisited node is 2; we can run DFS on that node, and since nodes 4 and 5 have already been visited, all that's left to do is push 2 on our final stack. The output of the algorithm is now 5-4-3-1-2, which is a valid ordering. In order to keep track of whether the topological ordering is possible (ie. there are no cycles), we can keep track of the current node that we're doing DFS on, and if a node we arrive at during traversal is the same as that starting node (aka we are in a cycle), then we can break out of the loop and return that there is no possible ordering. If there is a cycle, there cannot be a hierarchy of nodes, since a lower node will always redirect to a higher node.

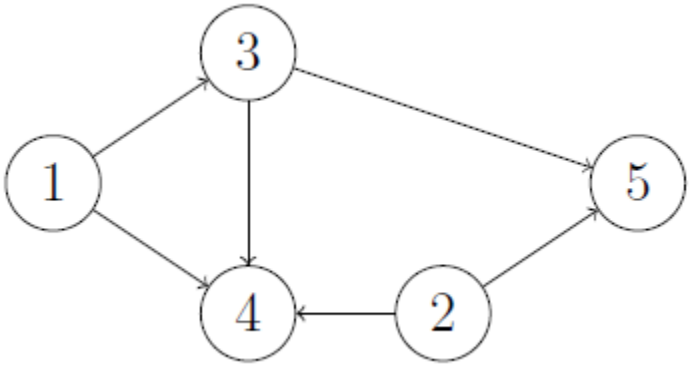

Figure 1: An acyclic, directed graph.

Pseudocode

```
def DFS_no_cycle(v, E):
    # does depth-first search, breaking if there's a cycle
    search = [v]
    visited = {v: True}
    if v has edges in E:
        if any of those edges are in visited:
            raise an error => There's a cycle!!!
        push connected nodes to search and add them as new keys in visited
        run DFS_no_cycle(connected nodes, E)
    return search


def top_order(V, E):
    # returns topological order of nodes
    visited = {}
    order = []
    for v in V:
        visited[v] = False
    for every unvisited v:
        visited[v] = True
        search = DFS_no_cycle(v, E)
        while search is not empty:
            i = search.pop()
            if i not in order:
                order.push(i)
    return order
```

2. Counting Components

See mku.py.

3. Probability Analysis

As can be seen in Figure 2, the minimum probability decreases exponentially as the number of nodes increases. This is because while there are more nodes that need to be connected, for every node added to the graph, there are n-1 new possible edges that could exist, so the probability that the graph is connected increases by more and more each time a new node is added.
See mku.py for full implementation.

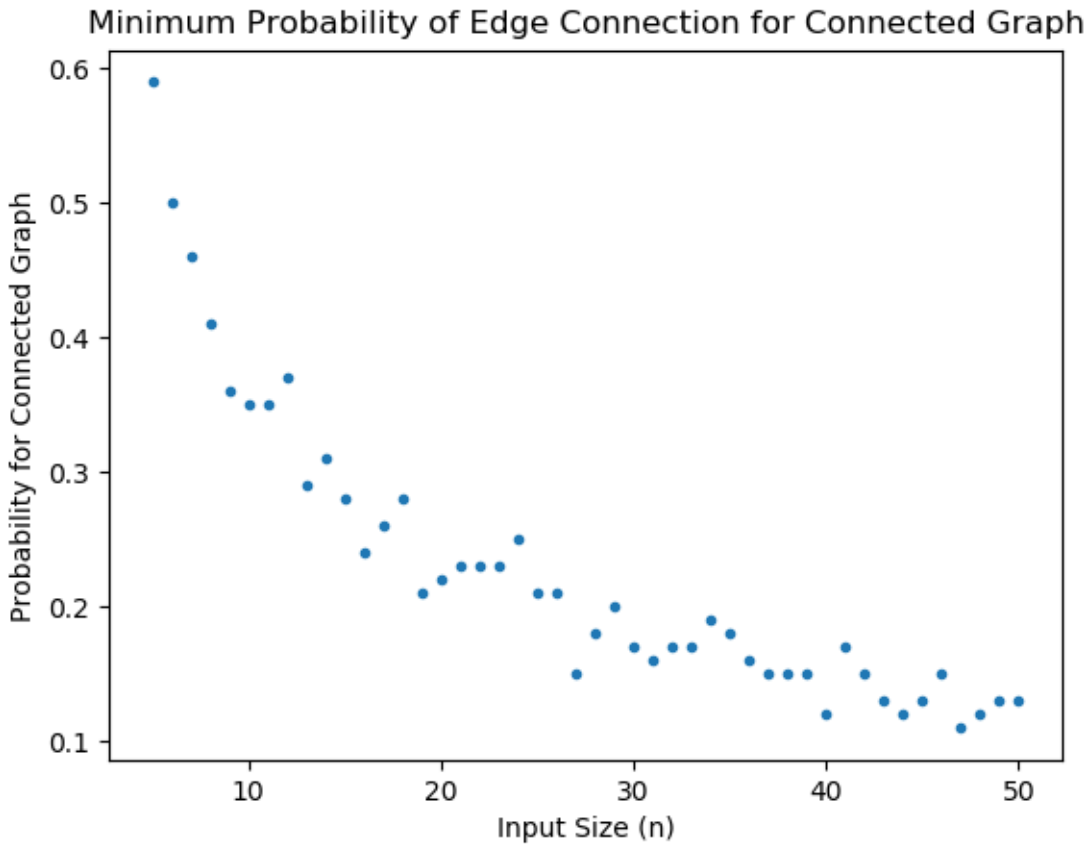

Figure 2: Graph of minimum probability, p, that results in connected graph.