Assignment 1:

Random Walk for Graph Connectivity Testing
SOFE 4820U Modelling & Simulation

Winter 2024

Daniel Gohara Kamel - 100754671

Jessica Leishman - 100747155

Design Decisions

Each separate component of the algorithm was contained within its own module to allow for easy modification and analysis of the individual components.

The graph is generated separately as an object, so that a list of nodes can be easily obtained and neighbors can be retrieved for it.

The random walk is conducted in a loop for a user specified number of times. It proceeds with the number of steps specified by the user, randomly selecting a neighbor from the current node. If there are no neighbors, the algorithm should terminate.

Connectivity is checked in its own module, as is optimal analysis.

Connectivity is simple to check, given the path produced by the random walk. If the path produced visits the same number of nodes as is contained in the starting graph then the graph is confirmed to be connected.

Testing the optimal amount of walks or steps is more difficult. As the nature of randomness states that we cannot guarantee that we will take the same amount of steps to visit each node each walk. There is also the matter of size, as the graph includes more nodes it becomes less likely that each node will be visited given the same amount of steps. The optimal amount of steps for a graph can be stated relative to the amount of nodes.

Code Analysis

The graph is generated by calling upon the random graph generation function provided by NetworkX. This allows for the random graph to have a user specified number of nodes, directed or undirected graphs, and disconnected graphs. A seed can also be included to allow for the reliable re-generation of the same graph, according to the NetworkX documentation. The function returns this graph object for use within the other components of the algorithm and analysis.

Random walk was split into two separate components, a loop to iterate over the random walk for the required number of times, and a separate implementation of the actual random walk algorithm. The start node is randomly selected from a list of nodes from the graph object using the random.choice function. Random walk is begun using the start node, for the number of steps specified by the user. If the node the walk is currently at has any neighbors, it selects one randomly from that list to proceed to and repeat until the number of steps is met. The neighbors function ensures that this procedure works for both directed and undirected graphs, or a walk that begins at a disconnected node.

Connectivity for an individual graph is assessed by conducting a random walk, and determining if all of the individual nodes from the graph list have been visited. If any have not, the graph is not connected. However, there is an optimal number of steps for the number of nodes in the graph to determine if the graph is connected while conducting a random walk, which is also completed as part of the analysis.

Optimal step analysis is conducted by assuming that the graph is connected, repeatedly random walking with increasing max step counts each time until a walk is confirmed to have visited every single node in the graph (or the user-defined MAX_STEPS is reached). Repeat this test an allotted “MAX_TESTS” amount of times to obtain an average step count to hit 100% of the nodes.

The above process is repeated with graphs of different node counts to obtain a trend line of how many steps are needed on average to confirm 100% connectivity as the amount of nodes in the graph increases.

Code Reflection and Improvements

The random walk algorithm used is easily modifiable through the use of user defined parameters and separate functions for each component of the algorithm. This allows for individual components to be easily modified to meet the requirements of the user.

The random walk algorithm may not be more efficient in terms of time or reliability. Where search algorithms like breadth first or depth first search can guarantee a result in a consistent amount of time, a random walk algorithm could theoretically, though highly unlikely for smaller more connected graphs, take an infinite amount of time and produce an inconclusive result.

Given random walk algorithm limitations for accessing general connectivity it may not be the best algorithm for this task. ST connectivity is another kind of connectivity that denotes if a certain two nodes S (start) and T (target) are connected. Random walk could solve this problem in a constant space complexity. Where the only thing checked is if the current node is the target node and the only space taken is a step counter to prevent infinite execution time.

Random walk could also be used to prove relatedness. By assigning each node a counter to track times visited, and repeating the random walk several times not clearing the counter each time, the result is now that each node has a score for how close or related it is to the starting node. Nodes that have more paths from the starting node, are more likely to be visited and have their counters incremented. This can be used for recommendation algorithms for things like movies on a streaming service or friend of friend recommendations on a social network.

Other improvements include steps to mitigate being trapped in cycles or random walking to nodes further away from unvisited nodes. This can be achieved by using random walk with restart. Random walk with restart is the same as regular random walk but each step has some constant, preset chance of teleporting to the initial node. This would be helpful on large tree-like graphs because as the walk progresses down one branch it would need to repeatedly, randomly, choose to move up the tree to reach branches that were not initially traversed. Tree structures often have each node with a single parent above and 2 or more children nodes below, making it more likely for the walk to traverse down the tree as opposed to above. As the walk progresses deeper down some branches it becomes increasingly unlikely that nodes on branches not chosen earlier will be reached. Restarting at the initial node provides another chance to select branches that were not traversed at a higher chance then randomly walking up the graph.