*

Homework 2 Written Problems

Jessica Saviano

Question 1:

For the graph G, represented by an adjacency matrix, in order to see if there is a node that has V -1 indegree and 0 out- degree, you have to run three separate loops (getting a runtime of O(3V), which is linear). First, it has to be clarified that the column for a specific node is the indegree, and the row is their outdegree. Therefore, if a whole column has 1's (meaning an edge is present), then that means every node will have an oudegree because there is at least one one in each row. Now beginning the algorithm, you start at the index [0][0] in the matrix and if the value is a 1, meaning there is an edge, you move down the matrix, and if its a 0 you move right across the matrix. If you are able to get all the way down the matrix, then that means the node we are looking for does not exist, because that means that every node as at least one out degree, where the node we are looking for has to have 0 out-degrees. If you are able to go all the way right in the matrix, that means you have a possible node that will satisfy our requirements, and you have to run another loop to make sure that that row you are on is all zeros. Then, you have to run your third and final loop to see if that column for that specific node is all 1's, except at the [i][i] spot (meaning where the node's column and row meet), and if that passes, the node is the one you are looking for! So, to summarize, to find a node that satisfies this you have to be able to get all the way right of your matrix, then at that specific row you end on you have to test to see if that row is all zeros and test to make sure that column (besides it's own column) is all 1's.

Question 2:

```
def dfs(graph, s):
    visited = []
    path = []
    return dfs_recurse(graph, s, t, visited ,path)

def dfs_recurse(graph, s, t, visited, path):
    path.append(s)
    visitied[s] = True
    if s == t:
        return path
    alist = graph.get_adjlist(s) //get all neighbors
    for v in alist:
        if v not in visited:
            dfs_recurse(graph,v,visited, path)
            if answer is not None: //we can go deeper in the branch
                return answer
    path.pop()
    if every element in visited is True:
        return []
    else:
        return dfs_recurse(graph,      return[]
```

Question 3:

A counter example can be given to show the claim that every node of an undirected connected graph G with n nodes has a degree of at least n/2 may not always be true. First, I will start off by defining what it means for an undirected graph to be connected, which is that every node in the graph must be accessible by another node in the graph. This means that for every node in the graph, you must be able to find some path from another node to get to that node. For the counter example of this claim, I will be describing a graph with n = 4 nodes, where each node does not have a degree of n/2 = 2 and the graph is still connected. One node in this graph can have a degree of 3, and we will call this node jess. The three other nodes in this graph is an edge with the node jess, therefore every other node in this graph can take a path through the node jess and get to every other node in the graph. Therefore, this graph is an undirected connected graph because every node can find a path to get to every other node, thus being a counterexample of the claim given.

Question 4:

For this problem, we want the robots to get to their destination in the shortest possible path without every interfering. So, our baseälgorithm would be breadth first search. We also have to think of this problem as, state -> space -> search -> setup, where each node is a particular state, and each edge is an action to get to each state. Both robots have a start state: s1 and s2, and an end state: d,d2, and In order to create a graph to run our algorithm on so that our robots can get to d1 and d1 we have to account for all the rules in the problem. The rules are that the robots cannot get to close to each other (same state or one state apart), and the robots must move one state at a time. So, you can only connect two states/nodes if they are only increasing the location of the robot by one, and if they aren't going to the same state nor going to states that are edges with each other. Now, since we have the graph created with the invalid edges accounted for, we can run our breadth first search algorithm to actually solve the problem. Breadth first search will calculate the shortest path for both robots while also making sure the paths are aligned with the constraints in the problem. The runtime for breadth first search is big theta V+E, where V is the number of states/nodes and E is the number of (valid) edges/edges in the graph. Breadth first search has this runtime because it will run through every node in the graph only one time, and going through each levelöf the graph, like peeling off layers on an onion. And, it will also go through every valid edge (therefore every edge in the graph) one time. As the number of robots grows, the runtime will also increase because there are more vertices and edges to account for and to run through, but ultimately the runtime will stay linear.