Jasmine B. McCrary

CSCI 406: Algorithms

February 3, 2024

Maze Project Report

1. Graph Model

In my graph model, each vertex represents a state. Each state is a possible position containing the combination of Lucky's or Rocket's current position. In my program, s1 represents Lucky's starting position and s2 represents Rocket's starting position. Hence, the start node represents a tuple in my code. My graph model also has a goal state. Each goal state has an edge to a single goal node.

The edges are directed and represent possible transitions between states. A transition is only possible if Lucky and Rocket can only move to the adjacent room through a corridor of the same color as the current room that the other person is in.

Overall, my model works by creating a graph based on the input nodes and edges, then creating a state graph that provides all the possible nodes that represent the rooms Lucky and Rocket can travel to, and the possible edges they can go through based on the condition. I have specified one main goal node, and if Lucky or Rocket reaches that node, they solve the maze.

2. Figure

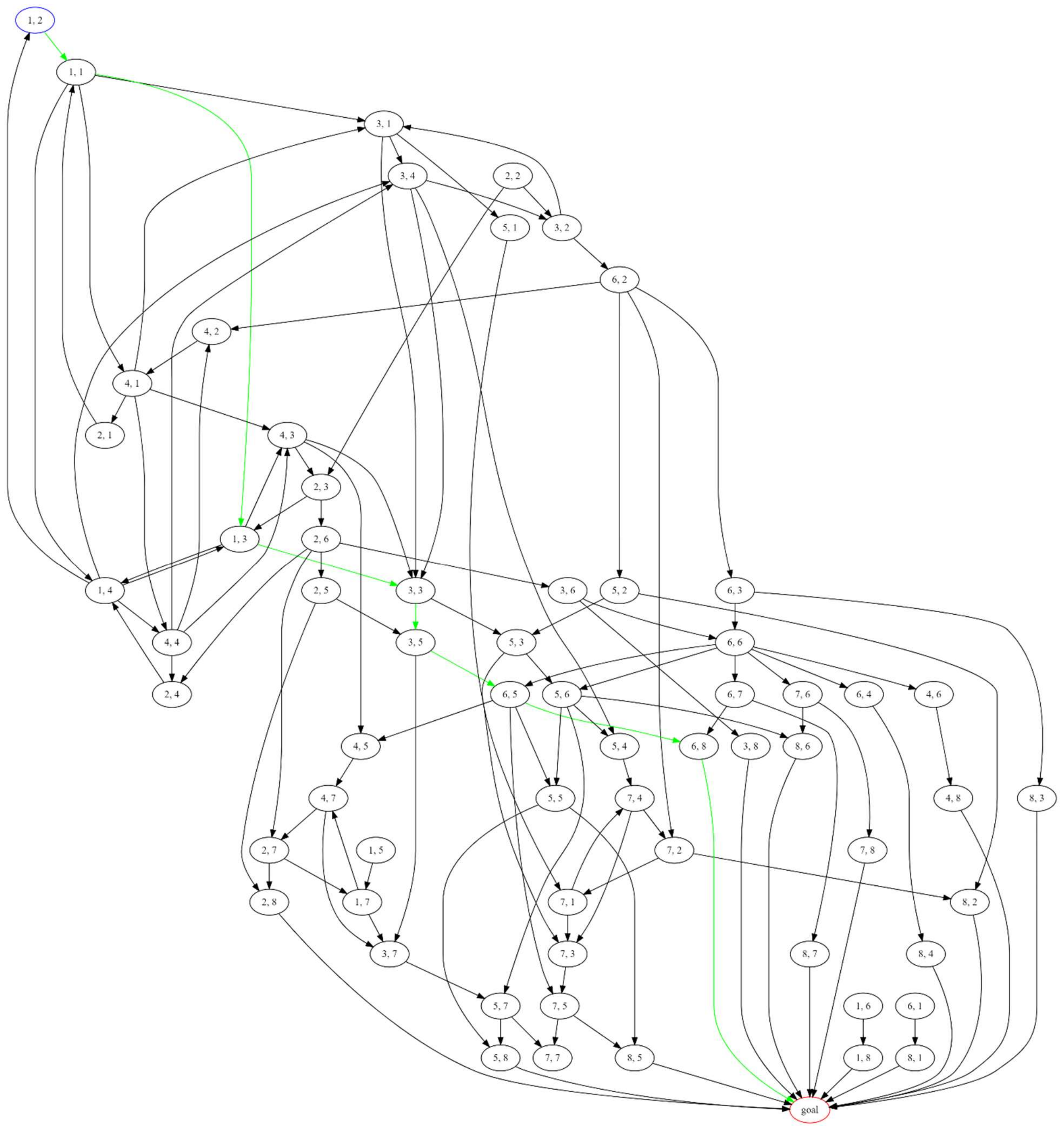

Figure 1: Graph that shows the shortest path highlighted by green where the start node is 1,2 (highlighted blue) and the end node is "goal" (highlighted red).

3. Correctness argument/proof

A graph model is a valid method to solve the problem if it can accurately represent all the possible nodes and edges depending on the conditions given and has the BFS algorithm perform the shortest path on it. Accordingly, my graph model follows this statement because it tracks what the start and final nodes are, it goes through each state, adds a node based on the position of the characters and adds all possible edges between the possible nodes as well. The main key points that my graph model focuses on are the start node, end node, the condition of when an edge will occur, the positions of Lucky and Rocket, and the colors of the room and the corridors. It also makes sure that the edges are in one direction. These criteria are crucial information in forming a graph for this problem, and my model considers all of this.

Since each vertex indicates the positions of Lucky and Rocket, its comprehensive representation of all possible states ensures that no scenario will be overlooked within the maze. Given that my model has a start and end node, it makes sure that the BFS algorithm can be run to find the shortest path.

The direction of the edges is also important, and my graph model encapsulates this rule, making sure that Lucky and Rocket’s movements are valid. The transitions only occur based on the condition of the corridor’s color matching the room color where the movement initiates.

In conclusion, my graph model accurately represents all possible states and possible transitions based on the condition given, runs the BFS algorithm and finds the shortest path.

4. Space Complexity

The precise formula for the number of vertices in relation to the size of the original maze is **n * n**. The tight upper bound for the number of edges is **O(n * m)** because, for every edge (m) and every node (n) that is read, there must be an edge made for Lucky and Rocket to transition to the next appropriate state. The worst-case scenario is when every room and edges have the same color as Lucky and Rocket will not be able to transition to the next state.

5. Code Submission

```python
#Jasmine B. McCrary

import sys
import networkx as nx

def inputForGraph(filename):
    with open(filename, 'r') as file:
        n, m = map(int, file.readline().split()) #read the num of nodes and num of edges and converts them to int
        roomColors = file.readline().split()#read the color of nodes
        s1, s2 = map(lambda x: int(x)-1, file.readline().split()) #read starting position for rocket and lucky
        edges = [tuple(line.strip().split()) for line in file.readlines()] #read edges from input
    startNode = (s1, s2) #initiate start node for the graph
    return n, m, roomColors, startNode, edges

def createGraph(n, roomColors, edges):
    g = nx.MultiDiGraph()

    #add node after reading from input
    for node in range(n):
        g.add_node(node, color=roomColors[node] if node < n - 1 else "W")
    #add the edges from input into the graph
    for first, second, color in edges:
        g.add_edge(int(first) - 1, int(second) - 1, color=color)

    return g
```

```python

def stateGraph(g, startNode, n):
    stateG = nx.DiGraph()

    #goal node
    goalN = 'goal'
    stateG.add_node(goalN)

    #implement all the states
    for s1 in range(n):
        for s2 in range(n):
            stateG.add_node((s1, s2))

            #if the possible node leads to goal node
            if s1 == n-1 or s2 == n-1:
                stateG.add_edge((s1, s2), goalN)

    #add possible edges between all possible nodes
    for possibleEdge in g.edges(keys=True):
        for possibleNode in g.nodes():
            #check the condition
            if g.edges(keys=True)[(possibleEdge)]['color'] == g.nodes()[possibleNode]['color']:
                stateG.add_edge((possibleEdge[0], possibleNode), (possibleEdge[1], possibleNode))
                stateG.add_edge((possibleNode, possibleEdge[0]), (possibleNode, possibleEdge[1]))

    return stateG, startNode, goalN

def BFS(stateG, startNode, goalN):
    everyPathSTR = []
    try:
        everyPath = nx.all_shortest_paths(stateG, startNode, goalN)

        for path in everyPath:
            outputSTR = ""
            for o in range(len(path)):
                if o < (len(path) - 2):
                    if path[o][0] != path[o + 1][0]:
                        outputSTR += "R" + str(path[o + 1][0] + 1)
                    if path[o][1] != path[o + 1][1]:
                        outputSTR += "L" + str(path[o + 1][1] + 1)
            everyPathSTR.append(outputSTR)
        return min(everyPathSTR)

    except nx.NetworkXNoPath:
        return "NO PATH"


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python script.py input.txt")
        sys.exit(1)
    filename = sys.argv[1]
    n, m, roomColors, startNode, edges = inputForGraph(filename)
    g = createGraph(n, roomColors, edges)
    stateG, startNode, goalN = stateGraph(g, startNode, n)

    shortestPath = BFS(stateG, startNode, goalN)
    print(shortestPath)

```

References

[1] "Python – Concatenate Strings in the Given Order," 10 May 2023. [Online]. Available: https://www.geeksforgeeks.org/python-concatenate-strings-in-the-given-order/.

[2] "Python Program for Breadth First Search or BFS for a Graph," GeeksForGeeks, 5 September 2023. [Online]. Available: https://www.geeksforgeeks.org/python-program-for-breadth-first-search-or-bfs-for-a-graph/.

[3] "Networkx - Documentation," NetworkX: Network Analysis in Python, [Online]. Available: https://networkx.org/documentation/stable/tutorial.html.