Duan Nguyen

CSCI 406

Section A

31st January 2024

Maze Report

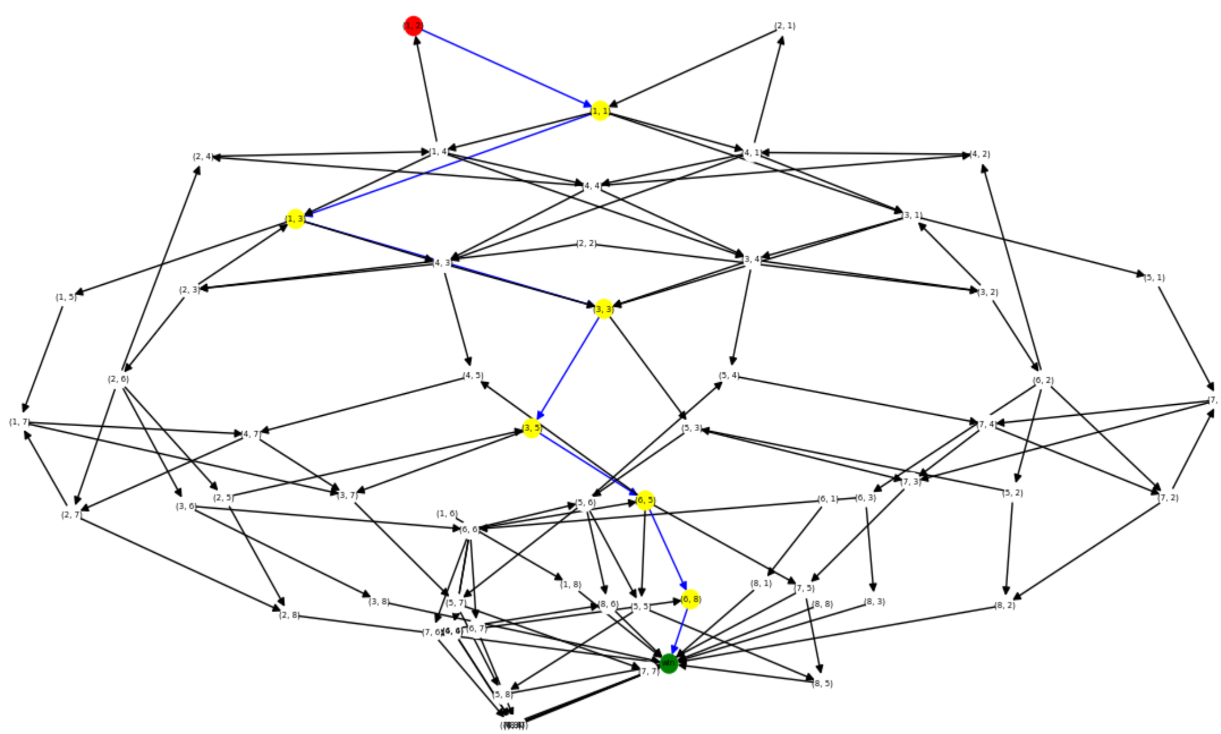
1. Logic Maze to Graph Model

    Vertices will be states containing a tuple of Rocket's and Lucky's current location on the original maze, and the starting state would be a tuple of where both Rocket and Lucky starts. The end state will be a state with a singular value of goal or win.

    Edges will connect one state to another state that are considered valid and possible, and a valid movement are when if and only if either Lucky or Rocket is in a certain color room and the same color corridor exist for only of them to move while the other stay in the said room. The game will end either Rocket or Lucky reach the final node, so any state that has at least one value of the final node in its tuple, that state will connect to the end node.

    Now the graph has only one starting state and one goal state, making traversing using BFS possible without modifying it.

2. Visualize Graph Model of Fig. 1



3. Correctness of Graph Model

    The nodes are now state with the coordinate pair of where Rocket and Lucky are instead of rooms of the logic maze, so each movement is changing between states instead of moving a pointer between rooms. The edges are now just directed connection from one state to another, and they only exist if the movements are possible in the logic maze. The starting location of both Rocket and Lucky is now a single state where the graph could start, and the goal state is now whenever either Rocket or Lucky reach the final node. So now the graph is simple directed graph with a single start and goal, BFS should be able to find the shortest path without modification, if possible, because if the logic maze is not possible, the state node would not have any connection to the goal state.

4. Space Complexity of Graph Model

    Given n as number of nodes for the maze, and m as the number of edges for the maze. For the logic map, the number of all possible nodes is calculated as n^2 since it is from tuples of nodes where both Rocket and Lucky are, so the space complexity of nodes is O(nn)

    From each given edge and node, 2 more edges can form for 4 possible states, 1 edge for 2 states of either Rocket stayed and Lucky moved and 1 edge for vice versa. So for every nodes and edges, the model can have up to 2nm edges, so the space complexity of edges is simplified to O(nm)

5. Code:

```
import networkx as net

def input_maze():
    # read in number of nodes and edges
    n_nodes, n_edges = input().split()
    n_nodes = int(n_nodes)
    n_edges = int(n_edges)

    # read in color map
    color_map = input().split()

    # read in Lucky and Rocket starting node
    rocket_start, lucky_start = input().split()
    rocket_start = int(rocket_start) - 1
    lucky_start  = int(lucky_start)  - 1
    starting_state = (rocket_start, lucky_start)

    # make maze
    in_maze = net.MultiDiGraph()
```

```
    # add nodes
    for node_num in range(len(color_map)):
        in_maze.add_node(node_num, color=color_map[node_num])
    in_maze.add_node(len(color_map), color="W")

    # read edges
    for i in range(n_edges):
        input_edge = input()
        # make sure for empty input
        if input_edge:
            from_node, to_node, edge_color = input_edge.split()
            in_maze.add_edge(int(from_node) - 1, int(to_node) - 1,
color=edge_color)

    # make state map
    state_graph = net.DiGraph()

    # win state
    win_state = 'win'
    state_graph.add_node(win_state)

    # add all states
    for lucky in range(len(color_map) + 1):
        for rocket in range(len(color_map) + 1):
            state_graph.add_node((rocket, lucky))

            # if state connect to winning
            if lucky == len(color_map) or rocket == len(color_map):
                state_graph.add_edge((rocket, lucky), win_state)

    # add edges between states
    for edge in in_maze.edges(keys=True):
        for node in in_maze.nodes():
            # color match
            if in_maze.edges(keys=True)[(edge)]['color'] ==
in_maze.nodes()[node]['color']:
                # add edges between either lucky colored room and rocket
movement and vice versa
                state_graph.add_edge((edge[0], node), (edge[1], node))
                state_graph.add_edge((node, edge[0]), (node, edge[1]))

    all_paths_str = []
    try:
```

```
        all_paths = net.all_shortest_paths(state_graph, starting_state,
win_state)

        for path in all_paths:
            out_str = ""
            for step in range(len(path)):
                if step < len(path) - 2:
                    # rocket moved
                    if path[step][0] != path[step+1][0]:
                        out_str = out_str + "R" + str(path[step+1][0] + 1)

                    # lucky moved
                    if path[step][1] != path[step+1][1]:
                        out_str = out_str + "L" + str(path[step+1][1] + 1)
            all_paths_str.append(out_str)

        print(min(all_paths_str))
    except net.NetworkXNoPath:
        print("NO PATH")
        return

input_maze()
```