Space Wreck Maze Problem

The space wreck maze problem has the basic following rules:

- The game has two players, Captain Rocket and Lieutenant Lucky.
- Each turn one player can move through a corridor to an adjacent room.
- The players can only move through corridors that are the same color as the room the other player is in.
- The game ends when either player reaches the goal room.

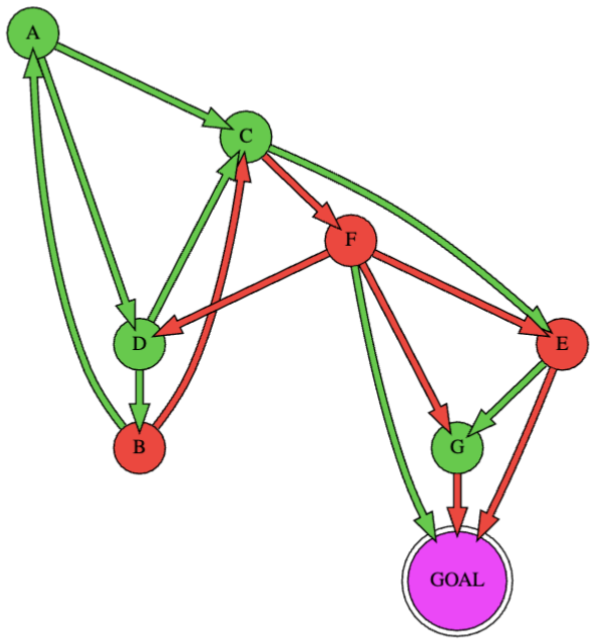


Figure 1: Sample Board Game (A=1,B=2...Goal=8)

Refer to Appendix A for the full problem description.

Graph Model

To determine the shortest path of moves for the SpaceWreck puzzle game, a Breadth-First Search (BFS) algorithm was applied to a graph of game states. The input file, containing room and corridor colors, served as the basis for generating the game state graph. A representative snapshot of the game state graph is depicted in Figure 2.

Nodes

The nodes in the game state graph correspond to each conceivable configuration of the game, identified by a tuple representing the room locations of Captain Rocket and Lieutenant Lucky. Figure 2 provides a visual representation of this graph.

Edges

The directed edges connecting nodes signify possible moves between game states. Each edge is labeled with the corresponding move required for the transition, denoting the player (Rocket or Lucky) and the new room number.

Special Features

To allow for an unmodified BFS search (i.e. one start vertex and one end vertex), a single node was generated to represent the end game state. When generating the explicit graph, any edge that would result in a single player in the goal room was connected to this single end node, represented at (n, n) on the state graph. This allows for a unmodified BFS search to be conducted from any game state node, representing the start, to the single goal state. Otherwise BFS must be modified to account for the $2n - 1$ unique end game states.

The explicit representation of game states and moves provides a comprehensive model for applying BFS to find the shortest path in the SpaceWreck puzzle game.

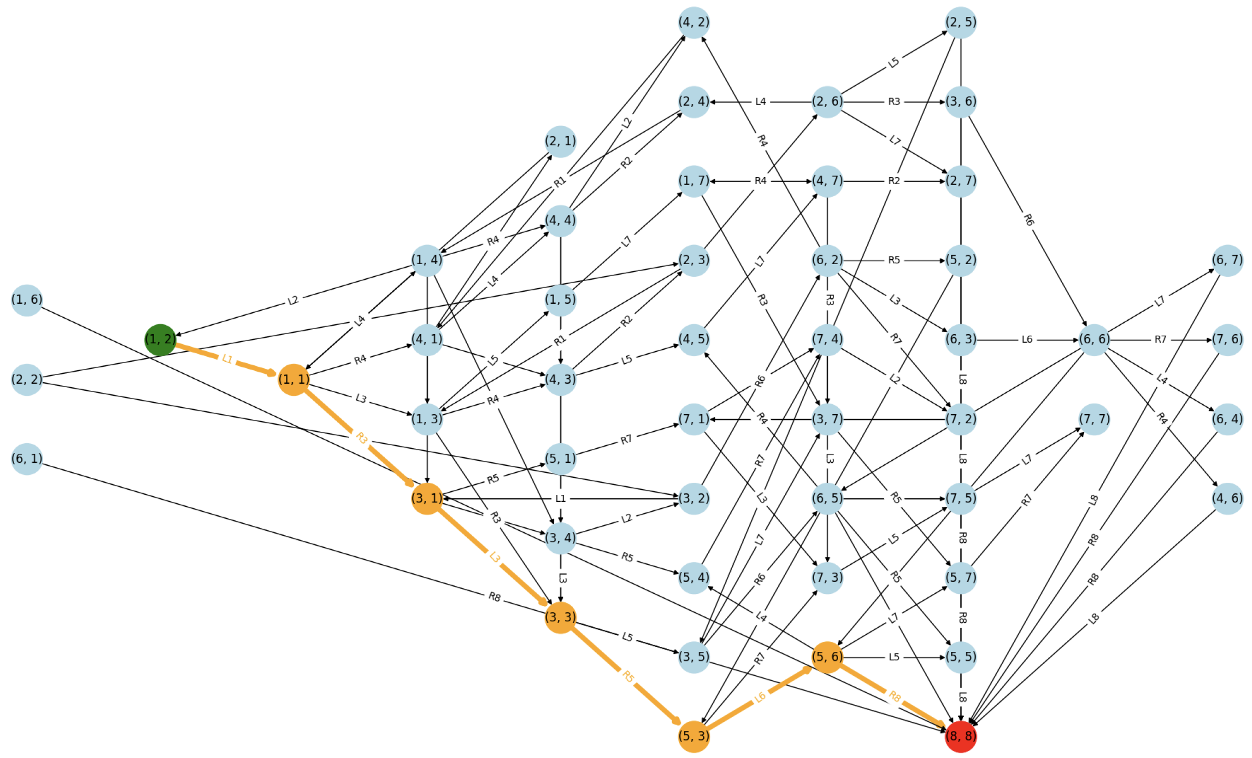


Figure 2: Game State Graph (green=start node, red=end node, orange=shortest path)

Model Accuracy

As discussed in the graph model section, each vertex on the game state graph represents a distinct configuration of the SpaceWreck puzzle. The edges between vertices correspond to valid moves between game states. Being a directed graph, if you select any vertex any children of that vertex are the valid moves from that game state.

As this graph is explicitly defined and represented the whole game state space, it is guaranteed to contain all possible configurations and moves within the maze. This ensures that every potential solution path is represented in the graph.

BFS was chosen as the search algorithm due to its ability to systematically explore the graph, starting from the initial game state and traversing through subsequent states level by level. BFS is guaranteed to find the shortest path, if one exist, when a directed graph has a finite branching factor and finite depth. This is the case for the SpaceWreck maze game state graph.

Space Complexity

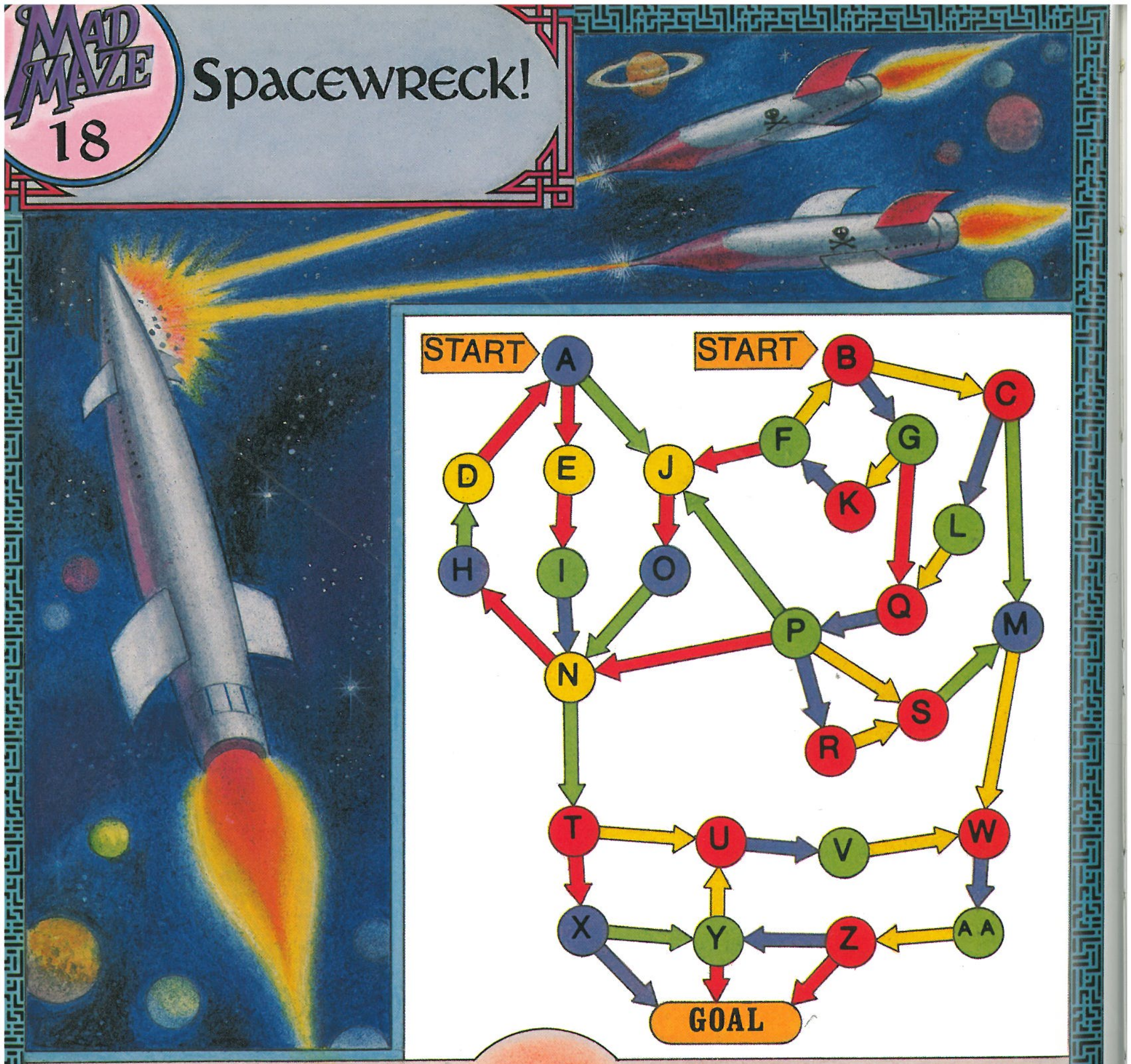
Assume a maze of n rooms and m corridors.

The total size of the generated game state graph is $(n-1)^2+1$ vertices[1]. The graph is initialized by generating a node for every possible combination of room locations for rocket and lucky, $(1 ... n-1, 1 ... n-1)$ and a single room of (n, n) is generated representing all possible game states which result in the end of the game.

The maximum number of edges in the state graph is $\frac{v(v-1)}{2}$, where $v=(n-1)^2+1$, which occurs when the state graph is fully connected. This is because each node has the potential to connect to every other node, except for itself. This results in the upper bound of $O(v^2)$ edges or $O(n^4)$ edges.

[1]the total number of nodes would be n^2 if the end node was not modeled as a singular node

1 Appendix A: Space Wreck Problem Description



The S.S. Fearless was on a routine spaceflight when it was attacked by space pirates. In a fierce battle, Captain Rocket and Lieutenant Lucky defeated the pirates, but their ship's Master Computer was damaged in the fight. Rocket is trapped in room A and Lucky trapped in room B. They cannot leave their rooms to enter the adjacent corridors because the life support system (normally controlled by the Master Computer) is now inoperative.

Lucky has a brilliant idea: use the manual override in each room to temporarily activate the corridor's life support system. Each room can activate the life support in any corridor of the same color. However, an operator must remain in the room while the corridor is being used. Each corridor is constructed for travel in only one direction, so whoever walks through a corridor must travel in the direction of the arrow. If either Rocket or Lucky can make it to the Master Computer, he can repair it; otherwise, they both will remain lost in space. How can one of them get to the Master Computer—at the point marked "Goal" on the maze?

Here's an example to show how they can move through the spaceship. Since Captain Rocket starts in a purple room, he could operate the controls to enable Lucky to travel along the purple corridor from room B to room G. Lucky would now be in a green room and he could operate the controls to let Rocket travel through the green corridor from room A to room J. Lucky could next move to K; Rocket could move to O; and Lucky could move to F. With Lucky now in a green room, Rocket could move down the green corridor to N; then he could make another move along the green corridor to T. (Note that you don't always have to alternate between Rocket and Lucky.) Captain Rocket is now getting close to the Master Computer and things are looking hopeful. But, alas, if you continue with this example you'll find that both space travelers will soon be trapped in endless loops. Whenever that happens, you'll just have to start over.

Appendix B: Python Code

```
##############################################
# Name: Maze Project
#
# The S.S. Fearless was on a routine spaceflight when it was attacked
    by space pirates. In a fierce battle, Captain Rocket and Lieutenant
    Lucky defeated the pirates, but their ship's Master Computer was
    damaged in the fight. Rocket is trapped in room A and Lucky trapped
    in room B. They cannot leave their rooms to enter the adjacent
    corridors because the life support system (normally controlled by
    the Master Computer) is now inoperative.
# Lucky has a brilliant idea: use the manual override in each room to
    temporarily activate the corridor's life support system. Each room
    can activate the life support in any corridor of the same color.
    However, an operator must remain in the room while the corridor is
    being used. Each corridor is constructed for travel in only one
    direction, so whoever walks through a corridor must travel in the
    direction of the arrow. If either Rocket or Lucky can make it to the
     Master Computer, he can repair it; otherwise, they both will remain
     lost in space. How can one of them get to the Master Computer—at
    the point marked "Goal" on the maze?
# Here's an example to show how they can move through the spaceship.
    Since Captain Rocket starts in a purple room, he could operate the
    controls to enable Lucky to travel along the purple corridor from
    room B to room G. Lucky would now be in a green room and he could
    operate the controls to let Rocket travel through the green corridor
     from room A to room J. Lucky could next move to K; Rocket could
    move to 0; and Lucky could move to F. With Lucky now in a green room
    , Rocket could move down the green corridor to N; then he could make
     another move along the green corridor to T. (Note that you don't
    always have to alternate between Rocket and Lucky.) Captain Rocket
    is now getting close to the Master Computer and things are looking
    hopeful. But, alas, if you continue with this example you'll find
    that both space travelers will soon be trapped in endless loops.
    Whenever that happens, you'll just have to start over.
#
# Author:  Brandon Ching
##############################################

# import matplotlib.pyplot as plt
import networkx as nx
import sys

##############################################
# Load the input file
```

```
##############################################

# if argument provided, get the input file from the argument
if len(sys.argv) > 1:
    input_file = sys.argv[1]
else:
    # output error message and exit if no input file is provided
    print("No input file provided")
    sys.exit()

# read the input file
with open(input_file, 'r') as f:
    lines = f.readlines()

# close the file
f.close()

##############################################
# Get basic information from the input file
##############################################

# Get the rooms and corridors from the first line
rooms = int(lines[0].split()[0])
corridors = int(lines[0].split()[1])

# Get the color of the rooms from the second line and convert it to a
   array of strings
room_colors = lines[1].split()
# append the goal room color to the end of the array
room_colors.append("goal")

# Get the starting rooms from the third line
captain_start = int(lines[2].split()[0])
lieutenant_start = int(lines[2].split()[1])

##############################################
# read the input file and create a graph of the game states
##############################################

# Create a graph of the game states
game_states = nx.DiGraph()

# create a node for every possible game state (each game state is a
   tuple of 2 rooms corresponding to the captain and lieutenant's
   positions)
for i in range(rooms-1, 0, -1):
    for j in range(rooms-1, 0, -1):
        game_states.add_node((i, j))

# create a single node for the goal state
game_states.add_node((rooms, rooms))

# read in the rest of the input file, where each line represents (
   origin room, destination room, color of the corridor)
for i in range(corridors):
    corridor = lines[i+3].split()
    origin = int(corridor[0])
    destination = int(corridor[1])
    color = corridor[2]

    # for each game state, check if either player is in the origin room
        and the other player is in a room with the same color as the
       corridor. If so, add an edge to the game state where the player
       in the origin room is in the destination room and the other
       player is in the same room as before
    for j in range(rooms, 0, -1):
        # Check if the captain is in the origin room and the lieutenant
            is in a room with the same color as the corridor
        if (origin, j) in game_states.nodes and color == room_colors[j
           -1]:
            # if destination is the goal room, add an edge to the goal
                state
            if destination == rooms:
                game_states.add_edge(
                    (origin, j), (rooms, rooms), key="R" + str(rooms))
            else:
                game_states.add_edge(
                    (origin, j), (destination, j), key="R" + str(
                        destination))

        # Check if the lieutenant is in the origin room and the captain
            is in a room with the same color as the corridor
        if (j, origin) in game_states.nodes and color == room_colors[j
           -1]:
            # if destination is the goal room, add an edge to the goal
                state
            if destination == rooms:
                game_states.add_edge(
                    (j, origin), (rooms, rooms), key="L" + str(rooms))
            else:
                game_states.add_edge(
                    (j, origin), (j, destination), key="L" + str(
                                destination))


############################################
# Process the graph to find the shortest path
############################################

# check if there are any paths from the starting game state to the goal
     state
if not nx.has_path(game_states, (captain_start, lieutenant_start), (
    rooms, rooms)):
     print("NO-PATH")
     sys.exit()

# find all shortest paths from the starting game state to the goal
    state
paths = nx.all_shortest_paths(
     game_states, (captain_start, lieutenant_start), (rooms, rooms),
         method="dijkstra")

# convert the paths to strings
output = []
for path in paths:
     path_output = ""
     for i in range(len(path)-1):
          path_output += game_states[path[i]][path[i+1]]["key"]
     output.append(path_output)

# sort all paths lexicographically and output the first one
output.sort()
print(output[0])


############################################
# Plot the graph
############################################

# # create a plot of the graph


def plot():
     # get the nodes of the shortest path
     shortest_path_nodes = nx.shortest_path(
          game_states, (captain_start, lieutenant_start), (rooms, rooms))
     color_map = []
     for node in game_states:
          if node == (captain_start, lieutenant_start):
              color_map.append('green')
          elif node == (rooms, rooms):
              color_map.append('red')
          elif node in shortest_path_nodes:
              color_map.append('orange')
          else:
              color_map.append('lightblue')

     # for each node, calculate the shortest number of steps to get from
          the start node to that node
     shortest_path = nx.single_source_shortest_path_length(
          game_states, (captain_start, lieutenant_start))

     # create a map of the distance from the start node to each node
         where the key is the distance and the value is a list of nodes
         with that distance
     node_distance = {}
     for node in shortest_path:
          if shortest_path[node] in node_distance:
              node_distance[shortest_path[node]].append(node)
          else:
              node_distance[shortest_path[node]] = [node]

     # if node is not in the map, add it to the map with a distance of
         -1
     for node in game_states:
          if node not in shortest_path:
              if -1 in node_distance:
                  node_distance[-1].append(node)
              else:
                  node_distance[-1] = [node]

     pos = {}
     # position the start node
     pos[(captain_start, lieutenant_start)] = (0, 0)
     # for each key in the node_distance map, position the nodes with
         that distance from the start node
     for key in node_distance:
          if key == 0:
              continue
          for i in range(len(node_distance[key])):
              pos[node_distance[key][i]] = (
                  key, (i - len(node_distance[key])/2) * 2)

     # make the edges connecting the nodes in the shortest path orange
```

```
        and thicker also make the font color of the edge labels orange
        and the rest of the edge labels black
    edge_color = []
    edge_width = []
    shortest_edge_labels = {}
    other_edge_labels = {}
    for edge in game_states.edges:
        if edge[0] in shortest_path_nodes and edge[1] in
            shortest_path_nodes:
            edge_color.append('orange')
            edge_width.append(5)
            shortest_edge_labels[edge] = game_states[edge[0]][edge[1]][
                "key"]
        else:
            edge_color.append('black')
            edge_width.append(1)
            other_edge_labels[edge] = game_states[edge[0]][edge[1]]["
                key"]

    # draw the graph
    nx.draw(game_states, pos, node_color=color_map, with_labels=True,
        node_size=1000,
            labels={node: node for node in game_states.nodes()},
                edge_color=edge_color, width=edge_width)
    nx.draw_networkx_edge_labels(
        game_states, pos, edge_labels=other_edge_labels, font_color="
            black")
    nx.draw_networkx_edge_labels(
        game_states, pos, edge_labels=shortest_edge_labels, font_color=
            "orange")

    plt.show()


# plot()
```