Ally Smith

Maze Project

1 Modelling

I modeled the maze using two separate graphs, one for forwards traversal and one for backwards. Each time the search lands on a circled node, it switches to the other graph, emulating the rules of Apollo's maze with Diana. Additionally, each node connects to the opposite color of nodes within the path of its direction. For example, a blue node facing northeast would connect to any red nodes along the NE diagonal from the original node in the forwards graph. In the backwards graph, the connections would be made with red nodes in the SW direction. Below is a small maze + the corresponding graphs:

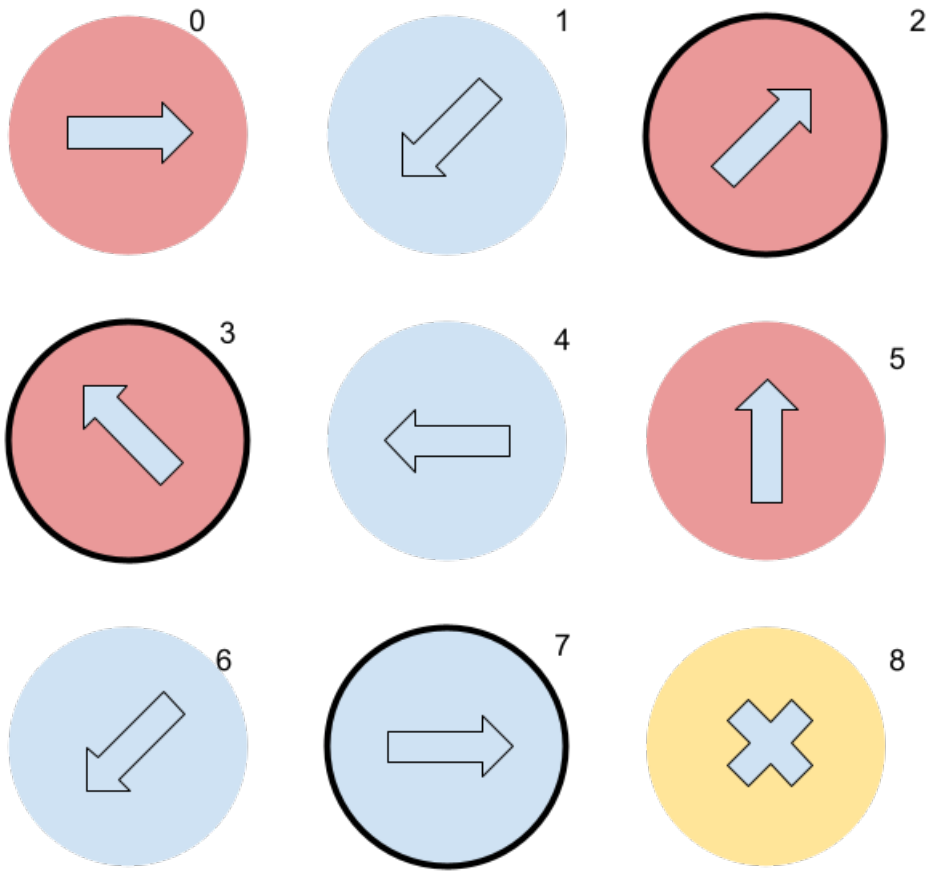

Figure 1: Example maze with a few circled nodes, each labeled with their index.

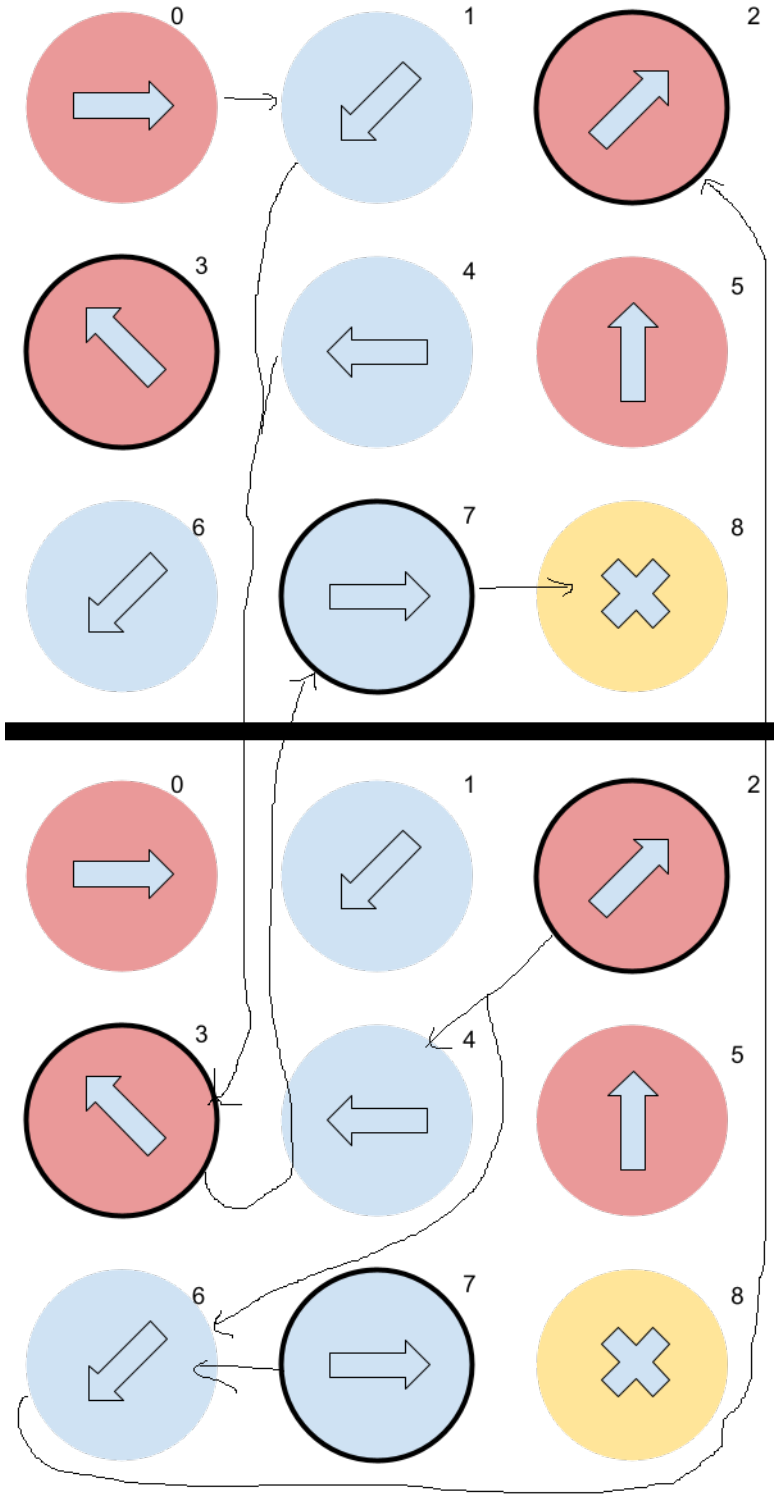

Figure 2: The graphs, recreated manually. The forward edges are on the top portion, and the backwards on the bottom. Lines that cross from top to bottom land on a circle and therefore must switch which part of the graph it is in.

Using this model for the problem, we can apply a breadth-first search algorithm to find the shortest path through the maze. We can convince ourselves this is the case by imagining this as one large graph rather than two graphs, as they are not disjoint. We simply draw lines from the circled nodes to the corresponding nodes in the opposite direction graph. This allows us to merge the two together into a single graph, and therefore BFS will always find a path through the maze if one exists.

2 Code

Below is the code that I used to work on this project, including the BFS algorithm I used to find the path through Apollo's maze. At the end of the pictures, you will see the output of my program

```
class Node:
    def __init__(self, dir: Direction, circ: bool, color: Color, pos: Tuple):
        self.direction = dir
        self.circled = circ
        self.row = pos[0]
        self.col = pos[1]
        self.color = color

    # string representation
    def __repr__(self) -> str:
        results = "[D: "
        if (self.direction == Direction.N): ...
        elif (self.direction == Direction.NE): ...
        elif (self.direction == Direction.E): ...
        elif (self.direction == Direction.SE): ...
        elif (self.direction == Direction.S): ...
        elif (self.direction == Direction.SW): ...
        elif (self.direction == Direction.W): ...
        elif (self.direction == Direction.NW): ...
        else: ...

        results += ", C?: "
        if (self.circled):
            results += "Y"
        else:
            results += "N"

        results += ", Col: "
        if (self.color == Color.RED):
            results += "R"
        elif self.color == Color.BLUE:
```

Figure 3: The basic node class I used to store the arrow & circle information.

```python
class Direction(Enum):
    N = 0
    NE = 1
    E = 2
    SE = 3
    S = 4
    SW = 5
    W = 6
    NW = 7


class Color(Enum):
    NONE = 0
    RED = 1
    BLUE = 2
```

Figure 4: Basic enumerations I used to store data in a more readable fashion.

```python
def print_grid(grid):
    for row in grid:
        for node in row:
            print(node, end="\t")
        print()


def BFS(id):
    visited = []
    queue = []

    queue.append(id)

    while queue:
        currentNode = queue.pop(0)
        visited.append(currentNode)
        print(f"Current node: {currentNode}")

        # determine if at bottom right corner, "exit"
        if max_r*max_c - 1 == abs(currentNode):
            return visited

        print(f"edges: {edges[currentNode]}")
        for neighbor in edges[currentNode]:
            if neighbor not in visited:
                queue.append(neighbor)

    return visited
```

Figure 5: Here is the BFS algorithm that I implemented.

```
grid, max_r, max_c = ri.read_file_to_2dlist("./input.txt")

print_grid(grid)
edges = ri.generate_edges_from_grid(grid)

visited = BFS(0)

# manually obtained from the file `sampleoutput.yml`
pathIDs = [0, 5, 29, 36, 43, 8, 22, 10, -38, -41, -13, -37, -45, -33, -25, -
           32, -11, -18, -39, 23, 25, 18, 4, 32, 39, 47, 7, -23, -21, -42, -24, 6, 48]

for nodeID in pathIDs:
    row = abs(nodeID) // max_r
    col = abs(nodeID) % max_c
    print(f"({row+1}, {col+1})", end=" ")
```

Figure 6: This is the code I used to drive my program, plus the code I used to calculate the coordinates in the appropriate output format for the assignment from the node IDs that I was using.

```
[D: E, C?: N, Col: R, (0, 0)]   [D: W, C?: N, Col: B, (0, 1)]   [D: NW, C?:
[D: SE, C?: N, Col: R, (1, 0)]  [D: S, C?: N, Col: B, (1, 1)]   [D: SE, C?:
[D: E, C?: N, Col: B, (2, 0)]   [D: N, C?: Y, Col: R, (2, 1)]   [D: S, C?: '
[D: N, C?: N, Col: R, (3, 0)]   [D: NE, C?: N, Col: R, (3, 1)]  [D: E, C?: '
[D: W, C?: N, Col: B, (4, 0)]   [D: S, C?: N, Col: R, (4, 1)]   [D: W, C?: '
[D: SW, C?: N, Col: R, (5, 0)]  [D: S, C?: N, Col: B, (5, 1)]   [D: NW, C?:
[D: SW, C?: N, Col: B, (6, 0)]  [D: N, C?: N, Col: R, (6, 1)]   [D: E, C?: I
Current node: 0
edges: [1, 2, 5, -6]
Current node: 1
edges: [0]
Current node: 2
edges: []
Current node: 5
edges: [11, 29, 35]
Current node: -6
{…
edges: []
Current node: 34
edges: []
Current node: 48
[0, 1, 2, 5, -6, 11, 29, 35, 36, 43, 8, -15, 22, -36, 10, -29, -22, 15, 24,
```

Figure 7: Above is the sample output of the program. It prints the grid, then prints each node as it gets to it and its neighbors. By working from the bottom up, you can find which node led you to where you are, and if you work all the way to the top you can determine the entire route. I identified my nodes starting at 0 up to $(r \times c) - 1$, as well as negative values for the backwards counterpart to each node, which can be seen at the bottom, which is the list of nodes in the order they were visited.

3 Results

When converted to the proper format, this is the output that I got:

(1, 1) (1, 6) (5, 2) (6, 2) (7, 2) (2, 2) (4, 2) (2, 4) (6, 4) (6, 7) (2, 7)
(6, 3) (7, 4) (5, 6) (4, 5) (5, 5) (2, 5) (3, 5) (6, 5) (4, 3) (4, 5) (3, 5)
(1, 5) (5, 5) (6, 5) (7, 6) (2, 1) (4, 3) (4, 1) (7, 1) (4, 4) (1, 7) (7, 7)