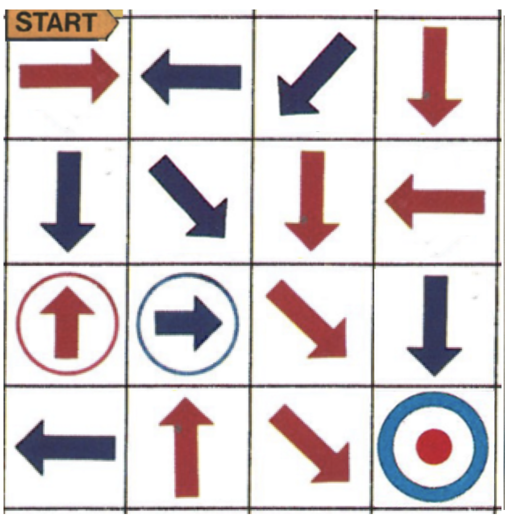
Maze Project

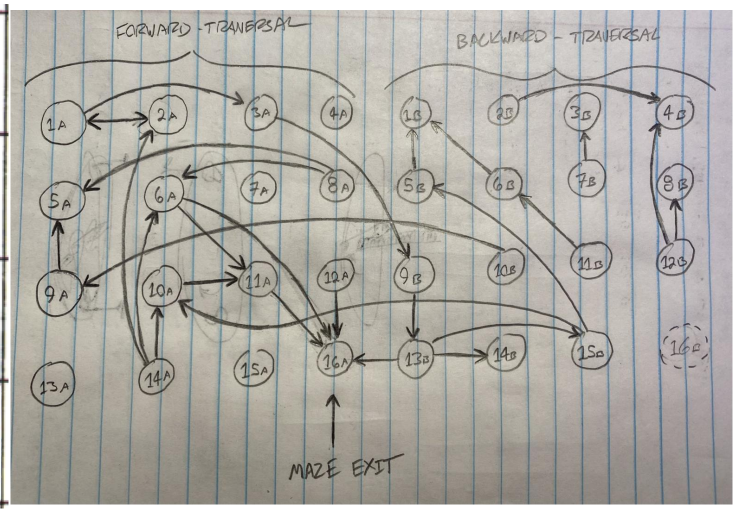
Liam Dempsey (10754465)

1.1. PROBLEM MODELING:

Graph Model: The graph that I used to represent the maze consists of two nodes for every location within the maze, one node representing the forward-traversal path of an arrow, and the other, the backward-traversal path. The theory behind this is that what matters when navigating a maze is from any location, what locations you can test next. Since the locations available to any maze location differ whether you are traveling forwards or backwards, effectively, this represents two distinct maze locations. The exception to this is the final "target" node, as traversal does not continue upon reaching it. However, for consistency, my implementation does technically contain two instances of this location as well, but only one is ever used.

Resulting Graph: I've opted to draw the entire graph of a smaller, simplified maze of my own creation. The following two images are the maze and its graph respectively.





Graph Algorithm: I have opted to use a breadth-first search to traverse the graph in search of a solution to the maze.

Algorithm Advocation: The breadth-first search algorithm will build a traversal tree of the graph starting from the initial node (namely (1, 1) in the case of this project) where every element is indexed by the ID of the node of interest, and whose value is the ID of the node that precedes the node of interest, when traversing from the first node. Because of the structure of this tree any element in this parent array with a value at its position, is guaranteed to have a path between itself and the initial node, which can be traced by recursively following the parent node "up" the graph. Furthermore, BFS will result in the shortest path to the target node. As the tree will not repopulate a value that already contains a parent node. Since this tree is built in a queue-like fashion, every iteration of the algorithm will be one level "deeper" in the search. As such, the tree will be built from relative least depth, to most depth.

1.2. PROGRAM CODE:

Program is written for Python 3.

Enum: Color, Direction

```python
from enum import Enum
import sys


# PROGRAM FLOW
# 1 : File IO - Parse input to array
# 2 : Graph creation - Transform formatted array into adjacency lists
# 3 : Search alg - Use DFS/BFS to convert the maze to a tree
# 4 : Solution - Pull a maze solution from the tree


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

    def __str__(self):
        return str(self.name)


class Direction(Enum):
    NONE = 0                # If none, node is the destination
    NORTH = 1
    NORTHEAST = 2
    EAST = 3
    SOUTHEAST = 4
    SOUTH = 5
    SOUTHWEST = 6
    WEST = 7
    NORTHWEST = 8

    def __str__(self):
        return str(self.name)


class Node:
    def __init__(self, id):
        self.id = id
        self.color = Color.NONE
        self.dir = Direction.NONE
        self.circled = False

    def __str__(self):
        return f"Node {self.id}: {self.dir}, {self.color}{' circled' if self.circled else ''}"

```

Class: Graph

```python
class Graph:
    def __init__(self, data):
        self.data = []      # Adjacency list (node is indexed ID - 1)
        self.tree = []      # Parent pointer array
        self.rows = 0
        self.cols = 0
        self.numnodes = -1
        self.build(data)

    def __str__(self):
        ret = ""
        for i, adj in enumerate(self.data):
            lst = ""
            for e in adj: lst += f"{((e - 1) % self.numnodes) + 1}{f'b, ' if e > self.numnodes else 'a, '}"
            ret += f"{(i % self.numnodes) + 1}{'a' if i < self.numnodes else 'b'}: {lst[:-2]}\n"
        return ret
```

```python
    def build(self, graph):
        self.rows = len(graph)
        self.cols = len(graph[0])
        self.numnodes = self.rows * self.cols

        # Build forward-facing nodes
        for row in graph:
            for node in row:
                if node.color == Color.NONE:
                    self.exit = node.id
                    self.data.append([])
                    continue

                x = 0      # Offset to search X
                y = 0      # Offset to search Y

                # 'Switch' for direction
                # Potential refactor: Store direction as N = [-1, 1], E = [-1, 1] instead of Enum type
                if node.dir == Direction.NORTH: y = -1
                elif node.dir == Direction.NORTHEAST: x, y = 1, -1
                elif node.dir == Direction.EAST: x = 1
                elif node.dir == Direction.SOUTHEAST: x, y = 1, 1
                elif node.dir == Direction.SOUTH: y = 1
                elif node.dir == Direction.SOUTHWEST: x, y = -1, 1
                elif node.dir == Direction.WEST: x = -1
                elif node.dir == Direction.NORTHWEST: x, y = -1, -1

                adj = []
                r = int((node.id - 1) / self.cols)
                c = node.id - (r * self.cols) - 1

                while not (x == 0 and y == 0):       # Loop until failure unless node is the target
                    try:
                        r += y
                        c += x
                        if r < 0 or c < 0: break

                        other = graph[r][c]
                        #print("OTHER:", other)

                        if node.color != other.color:
                            adj.append(other.id + (self.numnodes if other.circled else 0))

                    except IndexError:
                        break

                self.data.append(adj)

        # Build backwards-facing nodes
        for row in graph:
            for node in row:
                if node.color == Color.NONE:
                    self.data.append([])
                    continue

                x = 0      # Offset to search X
                y = 0      # Offset to search Y

                # 'Switch' for direction
                # Potential refactor: Store direction as N = [-1, 1], E = [-1, 1] instead of Enum type
                if node.dir == Direction.NORTH: y = 1
                elif node.dir == Direction.NORTHEAST: x, y = -1, 1
                elif node.dir == Direction.EAST: x = -1
                elif node.dir == Direction.SOUTHEAST: x, y = -1, -1
                elif node.dir == Direction.SOUTH: y = -1
                elif node.dir == Direction.SOUTHWEST: x, y = 1, -1
                elif node.dir == Direction.WEST: x = 1
                elif node.dir == Direction.NORTHWEST: x, y = 1, 1

                adj = []
                r = int((node.id - 1) / self.cols)
                c = node.id - (r * self.cols) - 1

                while not (x == 0 and y == 0):       # Loop until failure unless node is the target
                    try:
                        r += y
                        c += x
                        if r < 0 or c < 0: break

                        other = graph[r][c]
                        #print("OTHER:", other)

                        if node.color != other.color:
                            adj.append(other.id + (0 if (other.circled or other.color == Color.NONE) \
                                else self.numnodes))

                    except IndexError:
                        break

                self.data.append(adj)
```

```python
    def bfs(self, entry):    # Entry is int specifying node ID (converted to respective coords)
        self.tree = [None for i in range(self.numnodes * 2)]
        visited = [False for i in range(self.numnodes * 2)]
        queue = [entry - 1]

        while True:
            try:
                cur = queue.pop(0)
                adj = self.data[cur]
                visited[cur] = True
            except IndexError: break

            for node in adj:
                index = node - 1
                if not visited[index]:
                    queue.append(index)
                    self.tree[index] = cur + 1    # Build the tree

    def solve(self):
        solution = [self.exit]
        node = self.tree[self.exit - 1]

        while node is not None:
            solution.append(node)
            node = self.tree[node - 1]

        if len(solution) == 1: return "No solution exists"
        solution.reverse()

        ret = ""
        for step in solution:
            step = (step - 1) % self.numnodes
            ret += f"({int(step / self.cols) + 1},{(step % self.rows) + 1}) "

        return ret
```

File I/O parsing

```python
def parse(path):
    with open(path) as inf:
        raw = [line.split() for line in inf]

    rows = int(raw[0][0])
    cols = int(raw[0][1])

    # Empty table of data
    data = [[Node((x * rows) + y + 1) for y in range(cols)] for x in range(rows)]

    # Populate each node with its respective values
    for x in range(1, len(raw)):
        line = raw[x]
        node = data[int(line[0]) - 1][int(line[1]) - 1]

        color = line[2]
        circled = line[3]
        direction = line[4]

        # 'Switch' for node color
        if color == 'R': node.color = Color.RED
        elif color == 'B': node.color = Color.BLUE

        # 'Switch' for node direction
        if direction == "N": node.dir = Direction.NORTH
        elif direction == "NE": node.dir = Direction.NORTHEAST
        elif direction == "E":  node.dir = Direction.EAST
        elif direction == "SE": node.dir = Direction.SOUTHEAST
        elif direction == "S":  node.dir = Direction.SOUTH
        elif direction == "SW": node.dir = Direction.SOUTHWEST
        elif direction == "W":  node.dir = Direction.WEST
        elif direction == "NW": node.dir = Direction.NORTHWEST

        # Specify if node is circled
        if circled == 'C': node.circled = True

    return Graph(data)
```

Program main

```python
if __name__ == "__main__":
    path =  (sys.argv[1] if len(sys.argv) > 1 else None)
    if path is None:
        print("Usage maze.py {infile}")
        exit(-1)

    maze = parse(path)
    # print(graph)
    maze.bfs(1)
    solution = maze.solve()

    print(solution)
```

1.3. RESULTS:

The following is the program output when ran with the sample maze: Apollo’s Revenge

```
Drayuxs-MBP:P4 drayux$ python maze.py maze.dat
(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)
```