Maze Project Report

Nolan Donaldson

1.1 Problem Modeling

The maze was modeled as a graph by creating nodes for each space on the maze consisting of the coordinates of the space's position. There are two nodes for every space. The first node contains the space's coordinates, while the second node, the anti-node, contains the space's coordinates as negative numbers. The graph is created as a disconnected graph, with the set of normal nodes as one side and the set of anti-nodes as the other. Adjacency/target lists are created for each node, with normal nodes only having edges to other normal nodes and vice versa. However, each half of the graph is connected by the circle spaces on the maze, which are instead only given edges to the opposing half of the graph as shown by node (6,4) and node (-6,-4) in figure 1.

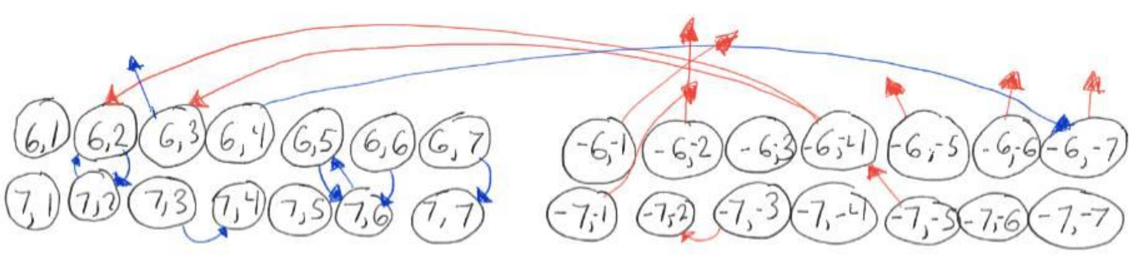

Figure 1. A visual representation of part of the graph.

In figure 1, node (6,4) is a circle, so its targets are only on the anti-half of the graph. This is also true for node (-6,-4), its corresponding anti-node, where its targets are contained to the normal side. Blue edges originate from the normal half and red edges originate from the anti-half of the graph; this colorization only helps for visual representation and is not reflected in the actual graph.

The searching algorithm used was BFS as it is optimal for finding the shortest possible path and doesn't waste resources with dead end paths as much as DFS.

BFS works on this graph because, while somewhat convoluted, this graph is not fundamentally different from any other directed graph. This maze requires some spaces to be used twice, which would normally be a problem on a graph containing only *n* (with n = the number of spaces on the maze) nodes. However, this graph contains *2n* nodes because of the existence of the normal/transpose distinction, so BFS only processes each node once. Once the maze has been completed, the parent of each node starting from the end space can be followed back to the start to get the shortest path from start to finish.

1.2 Code Submission

```python
# Author: Nolan Donaldson
# Maze project

# Used to store extra information during BFS search
class vertex:
    def __init__(self, color, parent):
        self.color = color
        self.parent = parent


# Acts as the 'vertices' for the maze layout
class Arrow:
    def __init__(self, row, column, color, circle, direction):
        super().__init__()
        self.row = row
        self.column = column
        self.color = color
        self.circle = circle
        self.direction = direction

# Creates adjacency list for each arrow
def build_graph(rows,cols,maze):
    # Tells the program how to translate depending on the direction of the arrow
    directions = {
        'N' : (0,-1),
        'E' : (1,0),
        'S' : (0,1),
        'W' : (-1,0),
        'NE': (1,-1),
        'SE': (1,1),
        'SW': (-1,1),
        'NW': (-1,-1),
        'X' : (rows,cols)
    }

    # Key: Each arrow | Value: list of the valid moves in the arrow's path
    # Graph consists of two distinct halves: the normal half and the
    # transpose half
    # These are connected by the circle arrow nodes
    graph = {}

    # Looks through the maze layout and creates the graph
    for currentRow in range(rows):
        for currentCol in range(cols):
            # Represents coordinates used to find each node's targets
            y,x = currentRow,currentCol

            # translateX, Y represent translations of x and y depending on the
            # arrow's head direction
            translateX,translateY = directions[maze[currentRow][currentCol].direc
tion]

            # node = tuple describing the location of each arrow
            # Coordinates are negative if it's a circle arrow, positive otherwise
            # This allows the circle arrows to have adjacency lists of nodes
            # inverse to themselves
            if maze[currentRow][currentCol].circle == 'C':
                node = (-(currentRow+1),-(currentCol+1))
            else:
                node = (currentRow+1,currentCol+1)

            # Creates key for node in the graph dictionary
            graph[node] = set()

            # Creates the graph, adding the nodes in the path of the current node
            # to that node's adjacency list (if they are the opposite color)
            while 0 <= y + translateY < rows and 0 <= x + translateX < cols:
                x += translateX
                y += translateY
                if maze[y][x].color != maze[currentRow][currentCol].color:
                    graph[node].add((y+1,x+1))

            # Reverses the node's direction (for the inverse nodes)
            if maze[currentRow][currentCol].circle == 'C':
                node = (currentRow+1,currentCol+1)
            else:
                node = (-(currentRow+1),-(currentCol+1))

            # Resets x and y
            y, x = currentRow,currentCol
            graph[node] = set()

            # Does the same as before, but for the nodes in the path of the
            # arrow's tail
            # Adds the nodes as negatives (to separate them from the normal ones)
            while 0 <= y - translateY < rows and 0 <= x - translateX < cols:
                x -= translateX
                y -= translateY
                if maze[y][x].color != maze[currentRow][currentCol].color:
                    graph[node].add((-(y+1),-(x+1)))
    return graph

# Function that reads in contents of the maze file
def initialize_graph():
    rows = 0
    columns = 0

    fMaze = open("maze.txt")

    # Reads in first line of the file to define size of the graph
    firstLine = fMaze.readline()
    firstLineInfo = firstLine.split(" ")

    rows = int(firstLineInfo[0])
    columns = int(firstLineInfo[1])

    # 2D list that contains each arrow node
    # Used for creating adjacency lists for the actual graph (a dictionary)
    mazeLayout = [[0 for i in range(columns)] for j in range(rows)]

    # Reads in the rest of the file into 2D list
    for i in range(rows):
        for j in range(columns):
            line = fMaze.readline()
            aspects = line.rstrip('\n').split(" ")
            arrow = Arrow(int(aspects[0]), int(aspects[1]), aspects[2], aspects[3
], aspects[4])
            mazeLayout[i][j] = arrow

    fMaze.close()
    return build_graph(rows, columns, mazeLayout)



# BFS for the maze
def BFS(graph):
    queue = []
    vertStore = {}

    # Initializes vertex attributes
    for arrow in graph:
        vertStore[arrow] = vertex('W', None)
```

```python
    # Adds start space to queue
    queue.append((1, 1))

    # Main BFS loop
    while len(queue) > 0:
        # Pops node from the queue
        u = queue.pop()
        # Checks each target of the node u
        for target in graph[u]:
            # Checks if the target node is undiscovered ('W')
            # If it is, add it to the queue and mark it as discovered ('G')
            # Also marks u as its parent node
            if vertStore[target].color == 'W':
                vertStore[target].color = 'G'
                vertStore[target].parent = u
                # If the target being looked at is the end, you're done
                if target == (7, 7):
                    return vertStore
                # Otherwise,
                queue.append(target)
        # Marks u as finished ('B')
        vertStore[u].color = 'B'



# Setup and maze search
graph = initialize_graph()
path = BFS(graph)

pathTaken = []

# Gets finish space
traveled = sorted(path.keys())[-1]

# Gets the path taken by the maze algorithm
while traveled != None:
    pathTaken.append(traveled)
    traveled = path[traveled].parent

# Follows the parent of each node starting from the end to get the path taken
for i in range(len(pathTaken)-1,-1,-1):
    print('(', abs(pathTaken[i][0]), ',', abs(pathTaken[i][1]), ')',sep='',end='
')
```

1.3 Results

(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)

Many of these nodes are anti-nodes with negative values, but only their absolute values are printed by the code. The results look like this using their real values:

(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)