Assignment 1 Solution

Jacob Bodera

2024-09-26

Problem 6

Implementation

For a more detailed view of the source code, please view the appendix at the bottom. The key positions include the starting position ("S"), ending position one ("1"), and ending position 2 ("2").

Breath-First Search For the breath-first search algorithm, I implemented an iterative approach using a first-in-first-out (FIFO) queue to store the list of open nodes to explore and a tuple which store the path taken to reach that node.

The algorithm begins by defining a list for the queue, with the starting position and path as its only entry, and a set to keep track of the nodes that have already been visited. The queue is now iterated over and the following steps are done:

- oldest entry from the queue is popped, with the x, y, and path values extracted into variables
  - if the position is equal to the goal, the function ends and returns the path taken to the goal node
- otherwise, each of the four possible directions are iterated over (chosen in the order right, down, left, up)
- if the new position is within the boundaries of the maze, not a wall, and not visited already, then the new position is added to the visited set and appended to the queue

This process is repeated until the goal is found or the queue runs out, meaning that no solution was found. Additionally, if a goal is found, the function prints the number of nodes visited and the cost of the path.

Depth-First Search For the depth-first search algorithm, I implemented a recursive approach using the direction order of right, down, left, then up. The path and visited are initially passed in as empty and a count variable is also passed in to keep track of the number of visited positions. The algorithm proceeds as following:

- test base case of the current position being equal to the goal position
- add the current position to the visited set and iterate the count by 1
- iterate through directions and recursively call function with new position, same end point, and the update path, visited, and count values
- if all positions are visited and no goal is found, then the function returns `None`

The function returns the path taken to reach the goal node and prints out the number of total positions visited and the cost of the path.

A* Search For the A* algorithm, I implemented an iterative algorithm, similar to that of the BFS described previously. This algorithm uses a heuristic function, which I chose to be the Manhattan Distance as it works very well on this style of problem which involves a 2D map where only horizontal and vertical moves are permitted. I also used a min heap which I used to efficiently remove the position with the lowest f(n) value. The algorithm proceeds as follows:

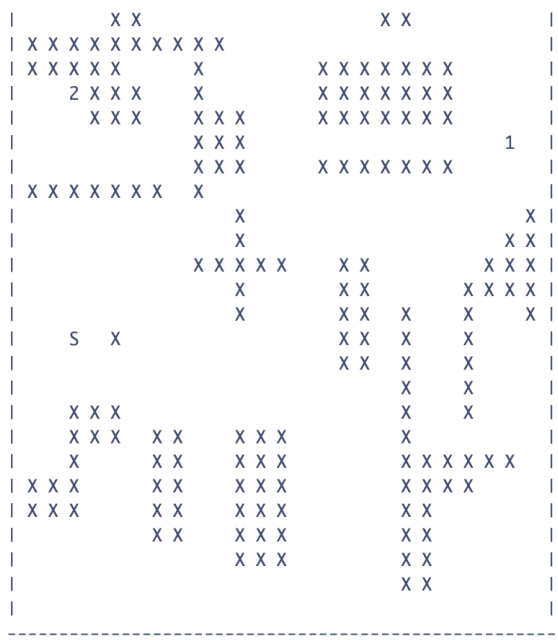
- create variables for the queue (initialized with the starting position), visited set, and a count to track the number of positions visited
- while the queue is not empty:
  - extract the g(n), position, and path from the min heap
  - if the position equals the goal, then return the path
  - loop through the four directions, calculate g(n) and f(n), mark position as visited, and add to the heap
- if there is no goal found, then the function returns `None`

This function returns the path taken from the start to the goal, as well as printing the number of positions visited and the cost of the path.

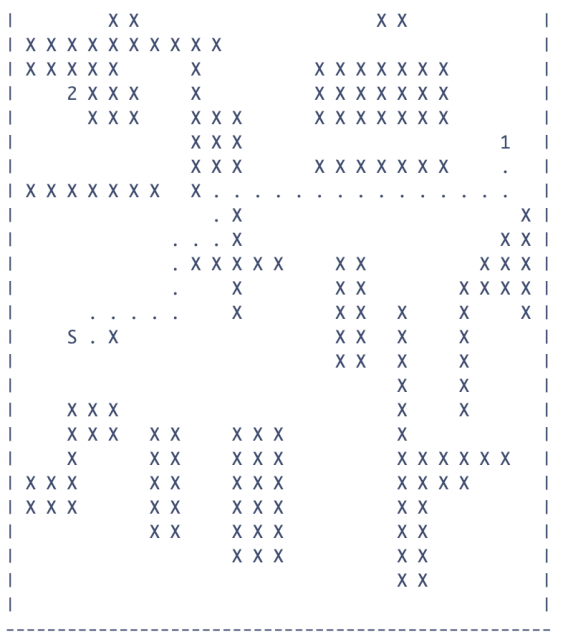
Testing Output

Output of Example Maze Layout

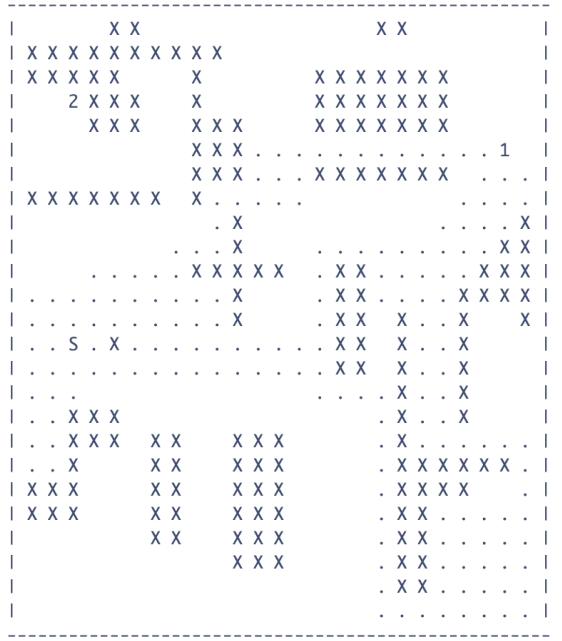
```
################## ORIGINAL MAZE #################
```



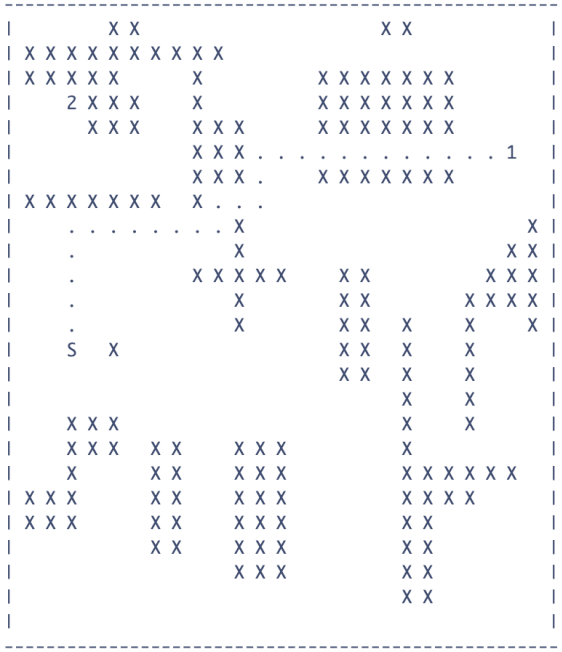
```
################## BREATH FIRST SEARCH ###############
Number of visited positions: 384
Cost: 30
```



```
################## DEPTH FIRST SEARCH ################
Number of visited positions: 178
Cost: 177
```

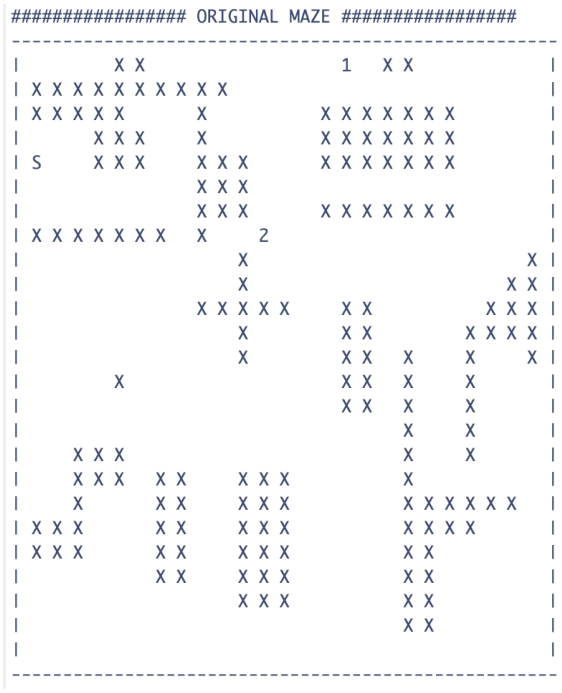


```
################## A* SEARCH #################
Number of visited positions: 76
Cost: 30
```

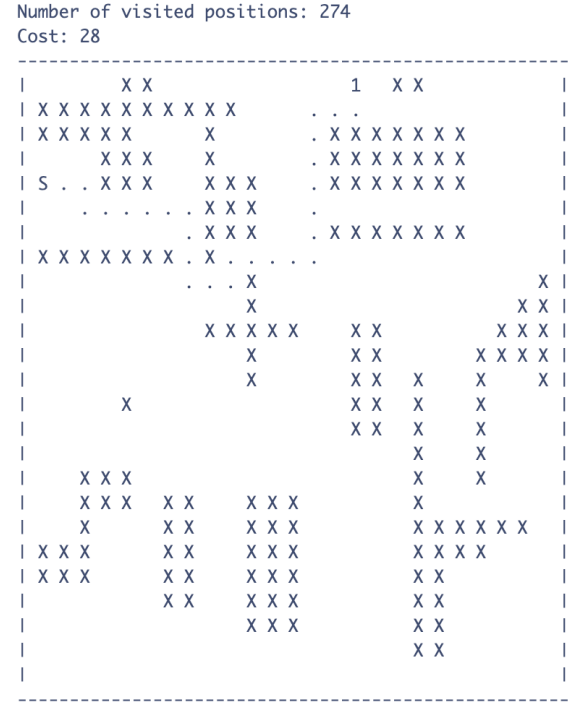


Output of BFS for Random Layout

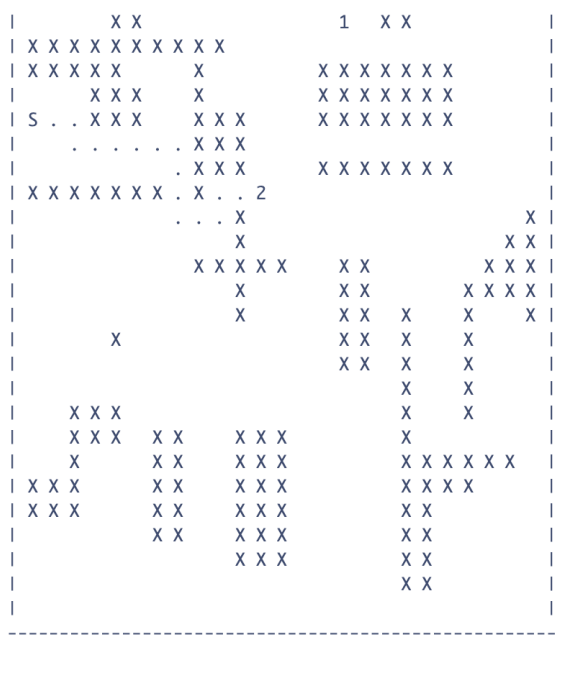
```
################## ORIGINAL MAZE #################
```



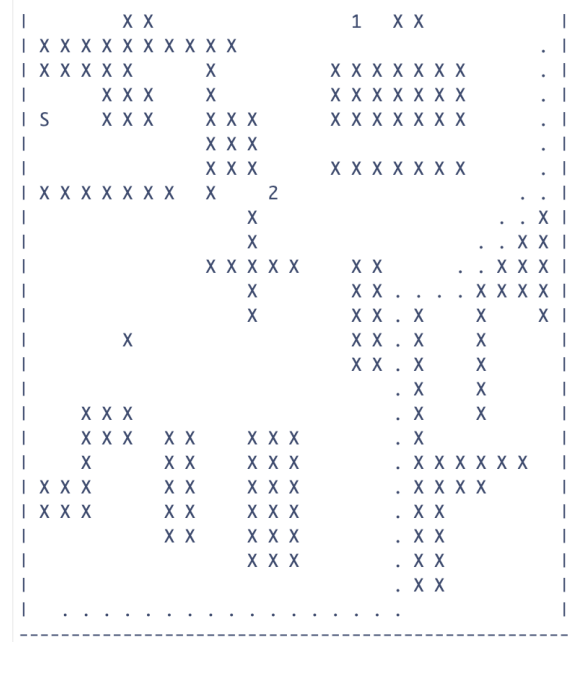
```
########### BREATH FIRST SEARCH (E1) ###########
Number of visited positions: 274
Cost: 28
```



```
########### BREATH FIRST SEARCH (E2) ###########
Number of visited positions: 61
Cost: 17
```

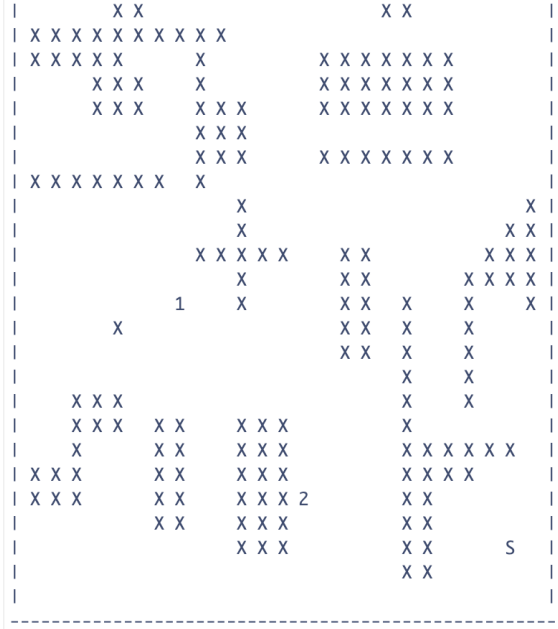


```
#### BREATH FIRST SEARCH (CORNER TO CORNER) ####
Number of visited positions: 448
Cost: 49
```



Output of DFS for Random Layout

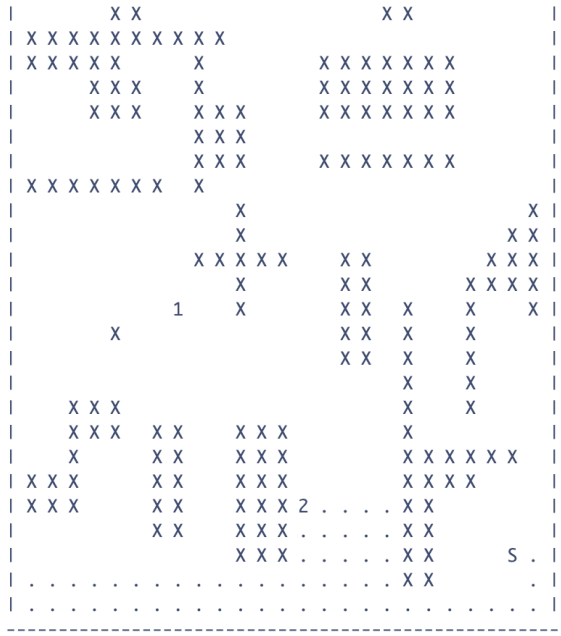
```
################# ORIGINAL MAZE #################
```



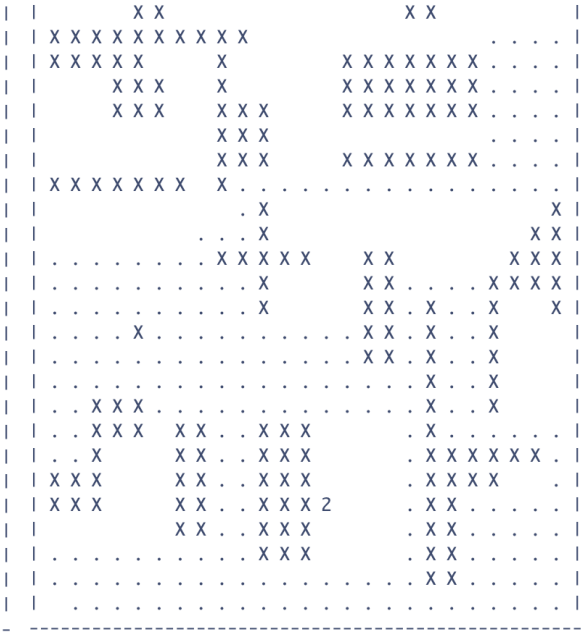
```
########### DEPTH FIRST SEARCH (E1) ###########
Number of visited positions: 240
Cost: 240
```



```
########### DEPTH FIRST SEARCH (E2) ###########
Number of visited positions: 60
Cost: 60
```

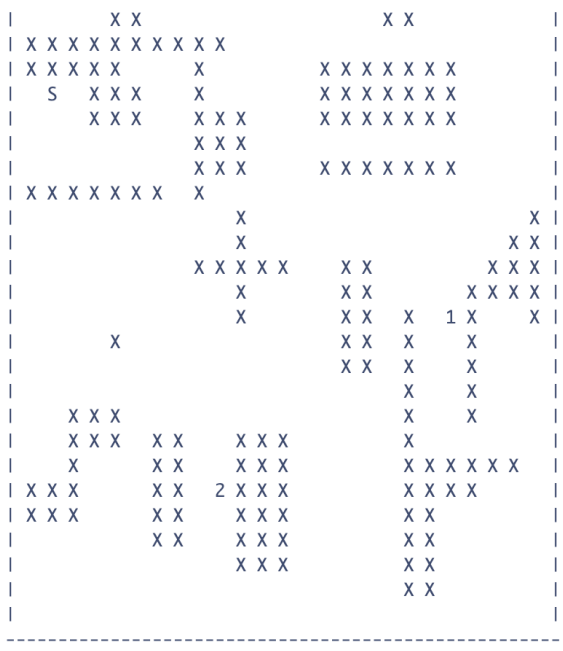


```
#### DEPTH FIRST SEARCH (CORNER TO CORNER) ####
Number of visited positions: 252
Cost: 252
```

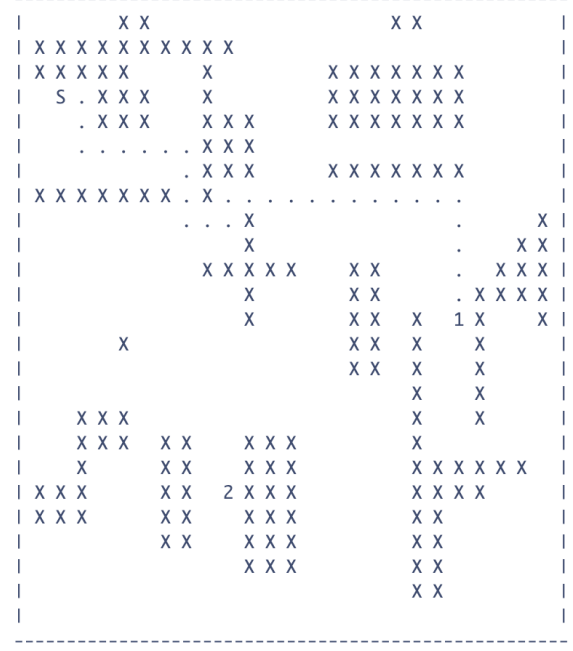


Output of A* for Random Layout

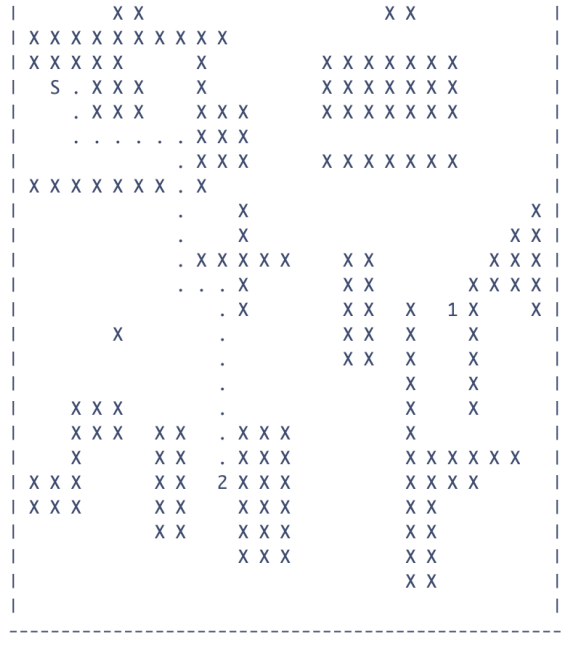
```
################ ORIGINAL MAZE #################
```



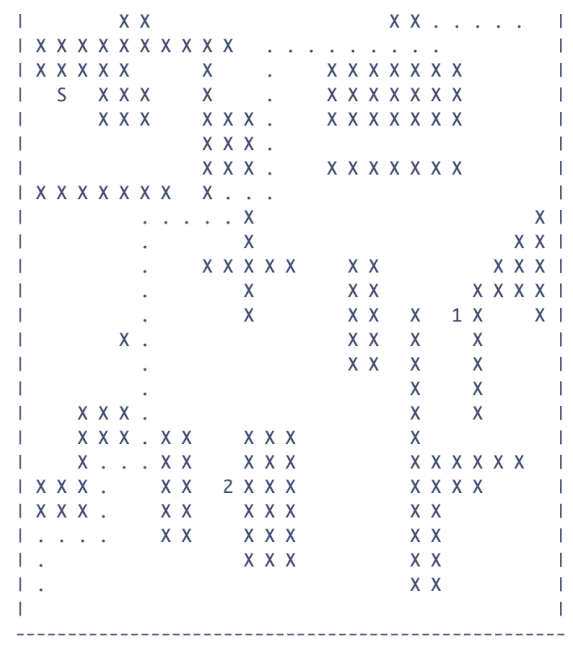
```
################ A* SEARCH (E1) ################
Number of visited positions: 102
Cost: 31
```



```
################ A* SEARCH (E2) ################
Number of visited positions: 49
Cost: 25
```



```
######### A* SEARCH (CORNER TO CORNER) #########
Number of visited positions: 339
Cost: 49
```



Appendix

mazeSearch.py

```
import heapq
import random
import copy

# heuristic function
def h_n(p1, p2):
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

def print_path(maze, path):
    m = copy.deepcopy(maze)
    path = path[1:len(path) - 1]
    for x, y in path:
        m[x][y] = "."

    print('-' * (len(m[0]) * 2 + 3))
    for i in range(0, len(m)):
        line = '| '
        for j in range(0, len(m)):
            line += m[i][j] + ' '
        print(line + "|")
    print('-' * (len(m[0]) * 2 + 3))

class MazeSearch:
    def __init__(self, start=None, end1=None, end2=None):
        self.maze = [[' '] * 25 for _ in range(25)]
        self.walls = [[4, 5, 17, 18],
                      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
                      [0, 1, 2, 3, 4, 8, 14, 15, 16, 17, 18, 19, 20],
                      [3, 4, 5, 8, 14, 15, 16, 17, 18, 19, 20],
                      [3, 4, 5, 8, 9, 10, 14, 15, 16, 17, 18, 19, 20],
                      [8, 9, 10],
                      [8, 9, 10, 14, 15, 16, 17, 18, 19, 20],
                      [0, 1, 2, 3, 4, 5, 6, 8],
                      [10, 24],
                      [10, 23, 24],
                      [8, 9, 10, 11, 12, 15, 16, 22, 23, 24],
                      [10, 15, 16, 21, 22, 23, 24],
                      [10, 15, 16, 18, 21, 24],
                      [4, 15, 16, 18, 21],
                      [15, 16, 18, 21],
                      [18, 21],
                      [2, 3, 4, 18, 21],
                      [2, 3, 4, 6, 7, 10, 11, 12, 18],
                      [2, 6, 7, 10, 11, 12, 18, 19, 20, 21, 22, 23],
                      [0, 1, 2, 6, 7, 10, 11, 12, 18, 19, 20, 21],
                      [0, 1, 2, 6, 7, 10, 11, 12, 18, 19],
                      [6, 7, 10, 11, 12, 18, 19],
                      [10, 11, 12, 18, 19],
                      [18, 19],
                      []]
        self.directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
```

```
    self.start = start
    self.end1 = end1
    self.end2 = end2

def set_walls(self, walls):
    self.walls = walls

def generate_maze(self):
    if self.walls is None:
        return self.maze

    for i in range(0, len(self.walls)):
        for j in self.walls[i]:
            self.maze[i][j] = 'X'
    if self.start is None or self.end1 is None or self.end2 is None:
        for i in range(0, 3):
            x = random.randint(0, len(self.walls) - 1)
            y = random.randint(0, len(self.walls) - 1)
            while self.maze[x][y] == 'X' or self.maze[x][y] == 'S' or self.maze[x][y] == '1':
                x = random.randint(0, len(self.walls) - 1)
                y = random.randint(0, len(self.walls) - 1)
            if i == 0:
                self.start = (x, y)
                self.maze[x][y] = 'S'
            if i == 1:
                self.end1 = (x, y)
                self.maze[x][y] = '1'
            if i == 2:
                self.end2 = (x, y)
                self.maze[x][y] = '2'
    else:
        self.maze[self.start[0]][self.start[1]] = 'S'
        self.maze[self.end1[0]][self.end1[1]] = '1'
        self.maze[self.end2[0]][self.end2[1]] = '2'

def print_maze(self):
    print('-' * (len(self.maze[0])*2+3))
    for i in range(0, len(self.maze)):
        line = '| '
        for j in range(0, len(self.maze)):
            line += self.maze[i][j] + ' '
        print(line + "|")
    print('-' * (len(self.maze[0])*2+3))

def bfs(self, start, end):
    queue = [(start, [start])]
    visited = {start}

    while queue:
        (x, y), path = queue.pop(0)

        if (x, y) == end:
            print("Number of visited positions: " + str(len(visited)))
            print("Cost: " + str(len(path)))
```

```
                return path

            for dx, dy in self.directions:
                nx, ny = x + dx, y + dy

                if 0 <= nx < len(self.maze) and 0 <= ny < len(self.maze[0]) and self.maze[nx][ny] != 'X
                    visited.add((nx, ny))
                    queue.append(((nx, ny), path + [(nx, ny)]))
        return None

    def dfs_recursive(self, x, y, end, path, visited, count=0):
        # Base case - goal is reached
        if (x, y) == end:
            print("Number of visited positions: " + str(count + 1))
            print("Cost: " + str(len(path)))
            return path + [(x, y)]

        # Mark current node as visited
        visited.add((x, y))
        count += 1

        for dx, dy in self.directions:
            nx, ny = x + dx, y + dy

            # Check if the new position is within bounds, not a wall, and not visited
            if 0 <= nx < len(self.maze) and 0 <= ny < len(self.maze[0]) and self.maze[nx][ny] != 'X' an
                result = self.dfs_recursive(nx, ny, end, path + [(x, y)], visited, count)
                if result:
                    return result

        # If no path is found from this node, return None (backtrack)
        return None

    def a_star(self, start, end):
        # ( f(n), g(n), (x, y), path )
        priority_queue = [(0 + h_n(start, end), 0, start, [start])]
        visited = set()

        while priority_queue:
            _, g_n, (x, y), path = heapq.heappop(priority_queue)

            if (x, y) == end:
                print("Number of visited positions: " + str(len(visited)))
                print("Cost: " + str(len(path)))
                return path

            visited.add((x, y))

            for dx, dy in self.directions:
                nx, ny = x + dx, y + dy

                if 0 <= nx < len(self.maze) and 0 <= ny < len(self.maze[0]) and self.maze[nx][ny] != 'X
                    new_g_n = g_n + 1
                    f_n = new_g_n + h_n((nx, ny), end)
```

```
                heapq.heappush(priority_queue, (f_n, new_g_n, (nx, ny), path + [(nx, ny)]))

    return None
```

problem6.py

```
from mazeSearch import *

### Generates output of example maze for each algorithm

'''
Maze in assignment:
start: (13, 2)
end1: (5, 23)
end2:  (3, 2)
'''

# Create maze
mz = MazeSearch(start=(13, 2), end1=(5, 23), end2=(3, 2))
mz.generate_maze()

# Print original maze
print("\n################## ORIGINAL MAZE #################")
mz.print_maze()

# Run breath first search with path shown
print("\n################## BREATH FIRST SEARCH #################")
bfs_path = mz.bfs(mz.start, mz.end1)
print_path(mz.maze, bfs_path)

# Run depth first search with path shown
print("\n################## DEPTH FIRST SEARCH #################")
dfs_path = mz.dfs_recursive(mz.start[0], mz.start[1], mz.end1, list(), set())
print_path(mz.maze, dfs_path)

# Run A* search with path shown
print("\n################## A* SEARCH #################")
a_star_path = mz.a_star(mz.start, mz.end1)
print_path(mz.maze, a_star_path)
```