CSB 310: Artificial Intelligence

Lab Assignment 4-5: Search Techniques

Submitted By:

Name: **Umang Kumar**

Roll No: **201210051**

Branch: **CSE**

Semester: **5th Sem**

Submitted To: ***Dr. Chandra Prakash***

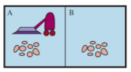
NATIONAL INSTITUTE OF TECHNOLOGY DELHI

Department of Computer Science and Engineering

2022

PART A : Introductory Problem [25 Marks]

Vacuum World [5 Marks]

1. Consider the vacuum world scenario discussed in class.  The only change is that instead of two rooms, there are four rooms (as shown below), one vacuum cleaner and a room may or may not contain dirt.

Room 1	Room 2
Room 3	Room 4

Inputs:
Write python/C program that takes four inputs through a single input file. First line of input file specifies the room number in which vacuum cleaner is located, second line specifies the presence or absence 1 or 0 of dirt in rooms 1, 2, 3 and 4, respectively, each separated by comma and third line in input file specifies the algorithm to be used :

dfs: Depth First Search
bfs: Breadth First Search

Assume valid actions as L,R,U,D,S,N where L=move_left, R=move_right, U=move_up, D=move_down, S=suck_dirt and N=no_op.

eg. In example in input file below (input.txt) vacuum cleaner is in Room 1, dirt is only in Rooms 1 and 4 and DFS is to be used in reaching to the goal state.
input.txt
1
1,0,0,1
dfs

CODE:

```
from tkinter import *
import time
from PIL import Image, ImageTk

root = Tk()

windowWidth = 600
windowHeight = 600

blockSize = 0
outlineLength = 0

root.geometry(f'{windowWidth}x{windowHeight}')

canvas = Canvas(root, width=windowWidth, height=windowHeight, bg="white")
canvas.pack()
```

```
img = Image.open('dirt.png')
vacuum = Image.open('vacuum.png')



class Cleaner_DFS:
   # if world[i][j] == 1 -> Dirt present
   # if world[i][j] == 0 -> noDirt present

   def __init__(self, world, vaccumClear_pos, rows, cols):
       self.world = world
       self.vaccumClear_pos = vaccumClear_pos
       self.covered = [[0 for x in range(0, cols)] for x in range(0, rows)]
       self.state = []
       self.dx = [-1, 1, 0, 0]
       self.dy = [0, 0, -1, 1]
       self.rows = rows
       self.cols = cols
       self.path = []
       self.roomsCovered = 0

       self.dfs(vaccumClear_pos)
       for i in self.path:
           print(i)

       time.sleep(1)
       root.destroy()

   def isAllCovered(self):
       return self.roomsCovered == (self.rows*self.cols)

   def isDirt(self, room):
       return self.world[room[0]][room[1]] == 1

   def get_roomNo(self, cur):
       X = cur[0]
       Y = cur[1]

       room = (X)*self.rows + (Y+1)
       return room

   def get_adjacent(self, X, Y):
       for k in range(0, 4):
```

```
            x = X + self.dx[k]
            y = Y + self.dy[k]

            if(x >= 0 and x < self.rows and y >= 0 and y < self.cols and
self.covered[x][y] == 0):
                direction = ''
                action = ''
                if k == 0:
                    direction = 'L'
                    action = "Moving Left"
                elif k == 1:
                    direction = 'R'
                    action = "Moving Right"
                elif k == 2:
                    direction = 'U'
                    action = "Moving Up"
                else:
                    direction = 'D'
                    action = "Moving Down"
                return [x, y, direction, action]

    def update(self, actions, cur_pos, next_pos):
        time.sleep(1)
        canvas.delete('all')

        for i in range(self.rows):
            for j in range(self.cols):
                canvas.create_rectangle(
                    i*blockSize, j*blockSize, (i+1)*blockSize, (j+1)*blockSize,
fill="white", outline="black")

                if world[i][j] == 1:
                    canvas.create_image(
                        i*blockSize, j*blockSize, anchor=NW, image=img)

        if self.isDirt(cur_pos):
            world[cur_pos[0]][cur_pos[1]] = 0

        mess = ""
        for i in actions:
            mess += i
            mess += '\n'
```

```
        canvas.create_text((cur_pos[0]+0.2)*blockSize, (cur_pos[1]+0.2)*blockSize,
                           text=mess, fill="black", font=('Helvetica', '15', 'bold'))

        obj = canvas.create_image(
            (cur_pos[0]+0.4)*(int(blockSize)), (cur_pos[1]+0.4)*(int(blockSize)),
anchor=NW, image=vacuum)
        # if next_pos != [-1, -1]:
        #     X = next_pos[0] - cur_pos[0]
        #     Y = next_pos[1] - cur_pos[1]
        #     k = 0
        #     while(k < 1):
        #         # time.sleep(1)
        #         canvas.move(obj, int((X*k)*blockSize), int((Y*k)*blockSize))
        #         k += 0.001

        root.update()

    def dfs(self, cur):
        self.roomsCovered += 1
        self.covered[cur[0]][cur[1]] = 1

        actions = []
        if self.isDirt(cur):
            actions.append("Dirt Found")
            actions.append("Suck Dirt")
            self.path.append([self.get_roomNo(cur), 'S'])
        else:
            actions.append("No Dirt found")
            self.path.append([self.get_roomNo(cur), 'N'])

        if self.isAllCovered():
            self.update(actions, cur, [-1, -1])
        else:
            x, y, direction, action = self.get_adjacent(cur[0], cur[1])
            actions.append(action)
            self.path.append([self.get_roomNo(cur), direction])
            self.update(actions, cur, [x, y])
            self.dfs([x, y])


class Cleaner_BFS:
    # if world[i][j] == 1 -> Dirt present
```

```
    # if world[i][j] == 0 -> noDirt present

    def __init__(self, world, vaccumClear_pos, rows, cols):
        self.world = world
        self.vaccumClear_pos = vaccumClear_pos
        self.covered = [[0 for x in range(0, cols)] for x in range(0, rows)]
        self.state = []
        self.dx = [-1, 1, 0, 0]
        self.dy = [0, 0, -1, 1]
        self.rows = rows
        self.cols = cols
        self.path = []
        self.roomsCovered = 0

        self.bfs(vaccumClear_pos)
        for i in self.path:
            print(i)

    def isAllCovered(self):
        return self.roomsCovered == (self.rows*self.cols)

    def isDirt(self, room):
        return self.world[room[0]][room[1]] == 1

    def get_roomNo(self, cur):
        X = cur[0]
        Y = cur[1]

        room = (X)*self.rows + (Y+1)
        return room

    def get_adjacent(self, X, Y):
        adjacent = []
        for k in range(0, 4):
            x = X + self.dx[k]
            y = Y + self.dy[k]

            if(x >= 0 and x < self.rows and y >= 0 and y < self.cols and
self.covered[x][y] == 0):
                direction = ''
                if k == 0:
                    direction = 'L'
```

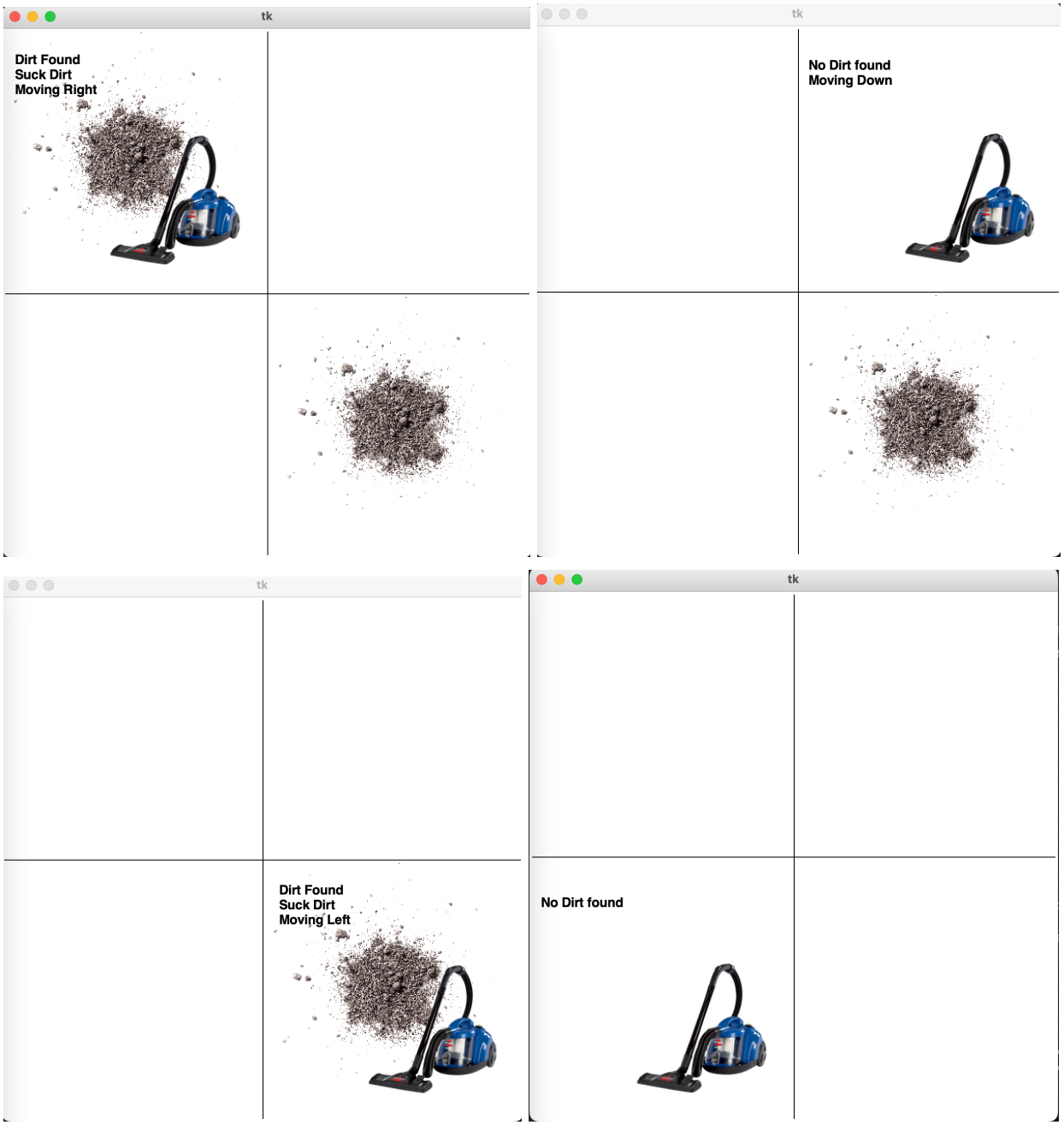
```
                elif k == 1:
                    direction = 'R'
                elif k == 2:
                    direction = 'U'
                else:
                    direction = 'D'
                adjacent.append([x, y, direction])

        return adjacent

    def next_directions(self, cur, next):
        X = next[0] - cur[0]
        Y = next[1] - cur[1]

        if self.dx[0] == X and self.dy[0] == Y:
            return 'L'
        elif self.dx[1] == X and self.dy[1] == Y:
            return 'R'
        elif self.dx[2] == X and self.dy[2] == Y:
            return 'U'
        else:
            return 'D'

    def bfs(self, start):

        queue = []

        self.covered[start[0]][start[1]] = 1
        queue.append([start])
        pre = [start]

        while len(queue):
            top = queue[0]
            queue.pop(0)

            # indx = len(pre)-1
            # while pre[indx] not in top:
            #     indx-

            if self.isDirt(top[-1]):
                self.path.append([self.get_roomNo(top[-1]), 'S'])
            else:
```

```
                self.path.append([self.get_roomNo(top[-1]), 'N'])

            for [x, y, direction] in self.get_adjacent(top[-1][0], top[-1][1]):
                if(self.covered[x][y]):
                    continue
                self.covered[x][y] = 1
                self.path.append(
                    [self.get_roomNo(top[-1]), self.next_directions(top[-1], [x, y])])
                queue.append(top + [[x, y]])


if __name__ == "__main__":
    pos = int(input())
    rows = 2
    cols = 2

    pos -= 1
    pos = [pos//rows, pos % rows]

    blockSize = min(windowWidth/rows, windowHeight/cols)
    outlineLength = blockSize/15
    img = img.resize((int(blockSize), int(blockSize)), Image.ANTIALIAS)
    img = ImageTk.PhotoImage(img)
    vacuum = vacuum.resize(
        (int(blockSize/2), int(blockSize/2)), Image.ANTIALIAS)
    vacuum = ImageTk.PhotoImage(vacuum)

    world = [[0 for x in range(0, cols)] for x in range(0, rows)]

    dirts = list(map(int, input().split(',')))
    for i in range(len(dirts)):
        r = (i)//rows
        c = (i) % rows
        world[r][c] = dirts[i]

    algo = input()

    if algo == 'dfs':
        def runCleaner():
            Cleaner_DFS(world, pos, rows, cols)

        root.after(1, runCleaner)
```

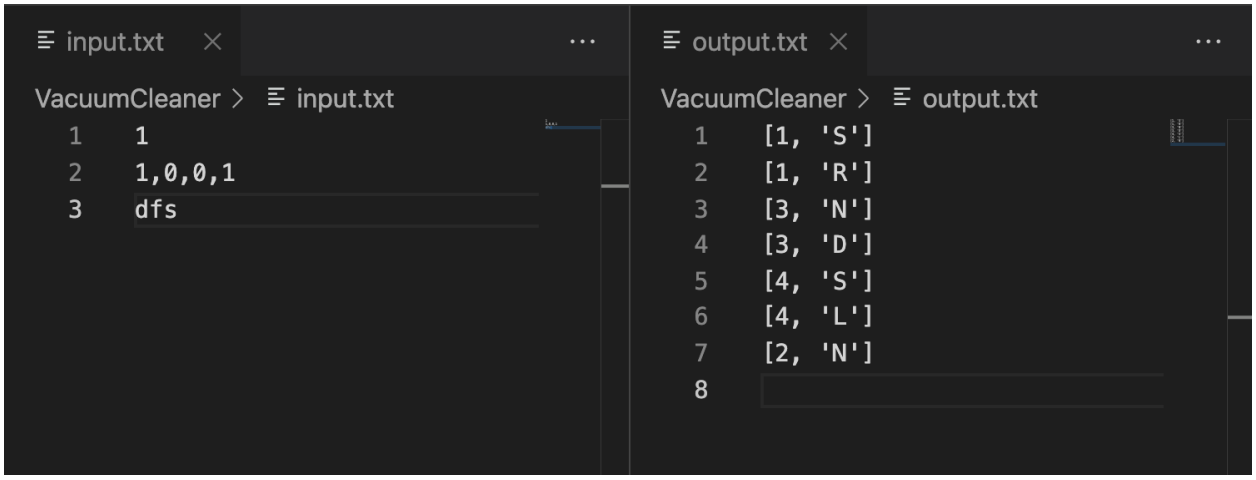
```
        root.mainloop()
    else:
        Cleaner_BFS(world, pos, rows, cols)
```

Output:

DFS-UI Output: (refer vacuum_dfs.gif in gifs folder)



Input/Output Format for DFS:

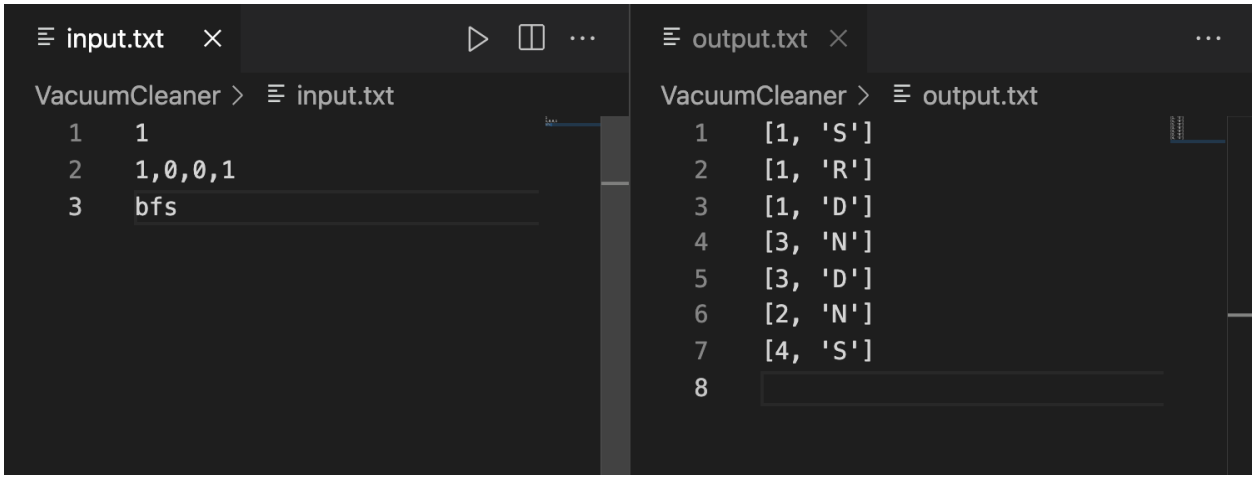


Explanation DFS:

DFS is depth first search. I have used the search techniques DFS in which we first completely explore one child of the node then likewise the second node and so on. In this problem I have used the tkinter library to design the GUI to demonstrate the working of DFS using the Vacuum Cleaner problem.

—-

Input/Output Format for BFS:



Explanation BFS:

BFS is Breadth First Search. In BFS search techniques we first explore all the children of the node then the childs of the children in the same fashion.

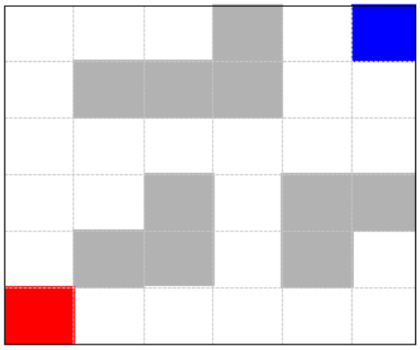
Observation:

- To demonstrate the working of DFS and BFS visually we have used tkinter and PIL library.

- The Vacuum Cleaner can suck the dirt, if dirt is present in the rain. The Vacuum Cleaner can move upward, downward, right or left rooms.
- In DFS, we first completely explore one child of the node then likewise the second node and so on.
- In BFS search techniques we first explore all the children of the node then the childs of the children in the same fashion.

Maze Problem [10 Marks]

2. Consider a maze comprising of square blocks in which intelligent agent can move either vertically or horizontally. Diagonal movement is not allowed. Cost of each move is 1.



Red block: initial position, **Blue block:** goal position and **Grey block:** obstacle

Apply following Blind/Uninformed and Informed algorithms :
(a) dfs: Depth first search
(b) bfs: Breadth first search
(c) dls: Depth limited search, use 3 as default depth
(d) ucs: Uniform cost Search
(e) gbfs: Greedy Best First Search
(f) astar: A* Algorithm

DFS:

CODE:

```
from tkinter import *
import time

root = Tk()

windowWidth = 600
windowHeight = 600

blockSize = 0
outlineLength = 0
```

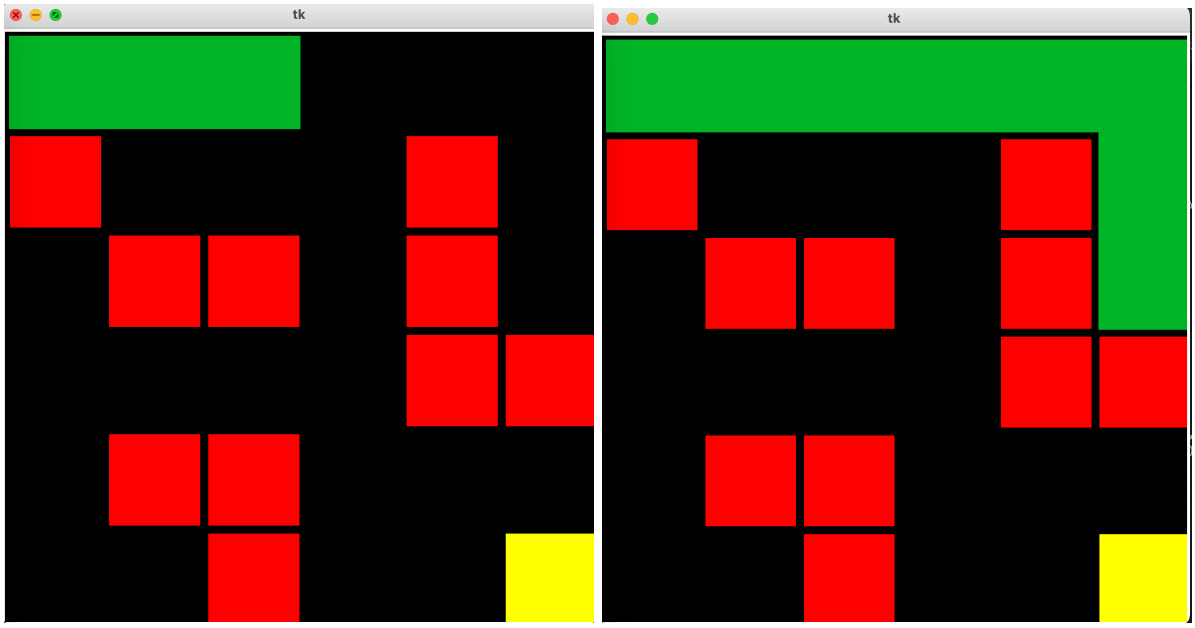
```
xOffset = 0
yOffset = 0

root.geometry(f'{windowWidth}x{windowHeight}')

canvas = Canvas(root, width=windowWidth, height=windowHeight, bg="black")
canvas.pack()


class Maze:

   def __init__(self, rows, cols, world, start, goal):
       self.reached = 0
       self.path = []
       self.dx = [-1, 1, 0, 0]
       self.dy = [0, 0, -1, 1]
       self.rows = rows
       self.cols = cols
       self.world = world
       self.start = start
       self.cost = 0
       self.goal = goal
       self.covered = [[0 for x in range(0, cols)] for x in range(0, rows)]

       self.dfs(start)
       if self.reached == 0:
           print("No path exist.")

       print("DFS: ", self.path)
       print(self.cost)

       root.destroy()

   # get all the possible adjacent nodes of the given coordinates
   def get_adjacent(self, X, Y):
       adjacent = []
       for k in range(0, 4):
           x = X + self.dx[k]
           y = Y + self.dy[k]

           if(x >= 0 and x < self.rows and y >= 0 and y < self.cols):
```

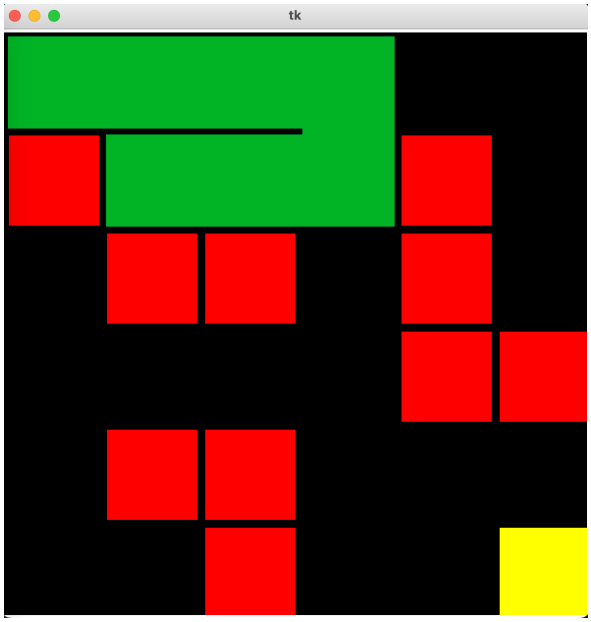
```
                adjacent.append([x, y])

        return adjacent

    def updateCanvas(self):
        time.sleep(0.1)
        canvas.delete('all')

        for i in range(rows):
            for j in range(cols):
                canvas.create_rectangle(
                    i*blockSize, j*blockSize, (i+1)*blockSize, (j+1)*blockSize,
fill="black")

        for i in range(rows):
            for j in range(cols):
                colorr = "black"
                if(world[i][j] == 2):
                    colorr = "red"
                elif world[i][j] == 3:
                    colorr = "yellow"
                canvas.create_rectangle(
                    i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill=colorr)

        for [i, j] in self.path:
            canvas.create_rectangle(
                i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill="#00B426", outline="#00B426")

        for x in range(1, len(self.path)):
            i = (self.path[x][0] + self.path[x-1][0])/2
            j = (self.path[x][1] + self.path[x-1][1])/2
            canvas.create_rectangle(
                i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill="#00B426", outline="#00B426")

        root.update()

    def dfs(self, cur):
        # if already visited or have reached goal return
        if self.reached == 1 or self.covered[cur[0]][cur[1]] == 1:
```

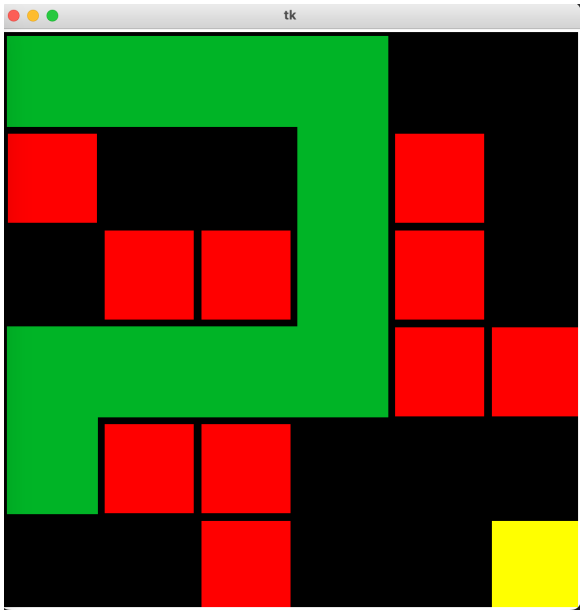
```
            return

        # push the cur node in the path and mark this node visited
        self.cost += 1
        self.path.append(cur)
        self.covered[cur[0]][cur[1]] = 1

        self.updateCanvas()

        # if current node is goal, mark reached as true
        if self.goal[0] == cur[0] and self.goal[1] == cur[1]:
            self.reached = 1
            return

        # traverse all the possible adjacent node of the current node
        for [x, y] in self.get_adjacent(cur[0], cur[1]):
            if(self.world[x][y] != 2):
                self.dfs([x, y])

        # if cannot reach to the goal node through this node then pop this node
        if self.reached == 0:
            self.path.pop()

        self.updateCanvas()


if __name__ == "__main__":
    reached = 0

    rows, cols = map(int, input().split(","))

    blockSize = min(windowWidth/rows, windowHeight/cols)
    outlineLength = blockSize/15

    xOffset = (windowWidth - blockSize*rows)/2
    yOffset = (windowHeight - cols*blockSize)/2

    start, end = input().split(";")
    start = list(map(int, start.split(',')))
    end = list(map(int, end.split(',')))

    start[0] -= 1
```

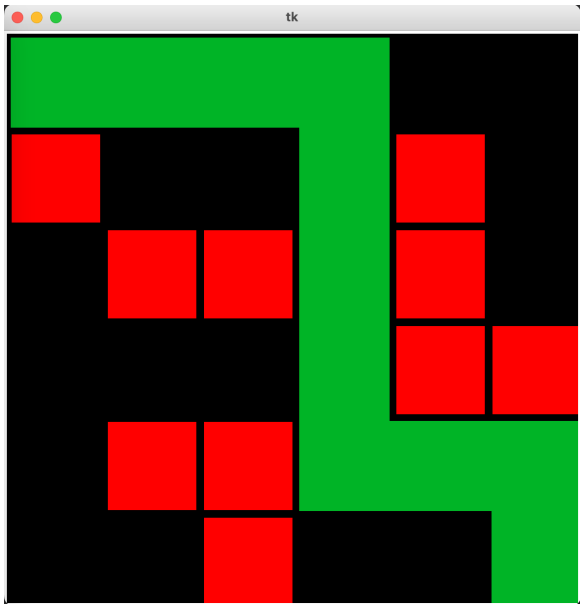
```
start[1] -= 1
end[0] -= 1
end[1] -= 1

world = [[1 for x in range(0, cols)] for x in range(0, rows)]
world[0][0] = 0
world[end[0]][end[1]] = 3

obstacles = list(input().split(";"))
for i in obstacles:
    x, y = map(int, i.split(','))
    world[y-1][x-1] = 2

def runMaze():
    Maze(rows, cols, world, start, end)

root.after(1, runMaze)
root.mainloop()
```

DFS-UI Output (refer maze_dfs.gif in gifs folder)

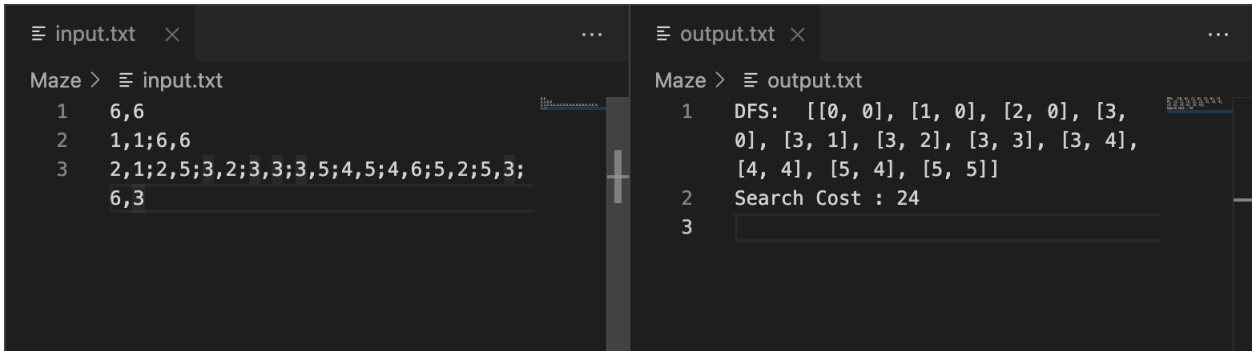



tk


tk


tk

Terminal Output:



———

BFS:

Code:

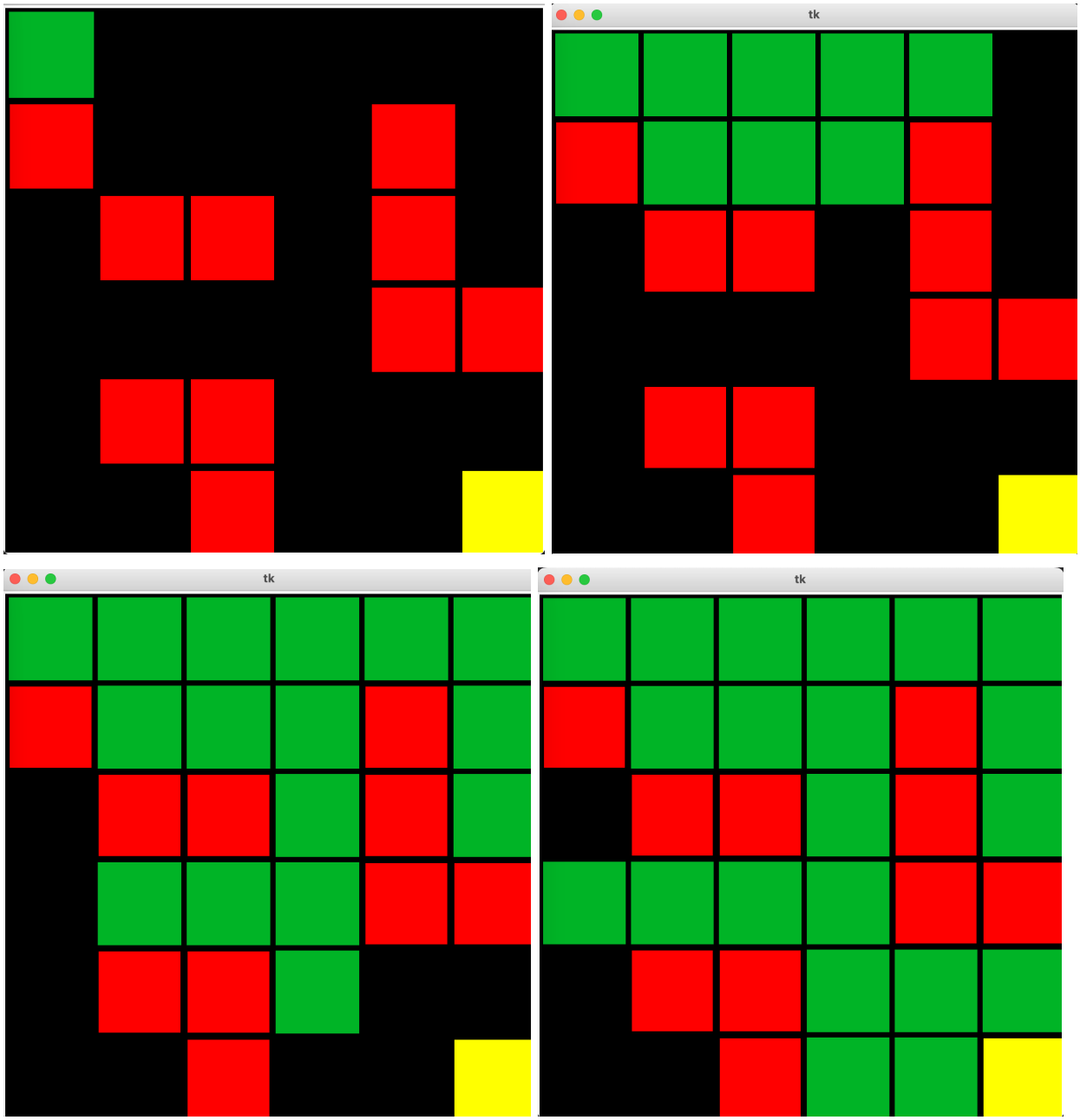
```
from tkinter import *
import time

root = Tk()

windowWidth = 600
windowHeight = 600

blockSize = 0
outlineLength = 0

xOffset = 0
yOffset = 0

root.geometry(f'{windowWidth}x{windowHeight}')

canvas = Canvas(root, width=windowWidth, height=windowHeight, bg="black")
canvas.pack()


class Maze:

   def __init__(self, rows, cols, world, start, goal):
       self.reached = 0
       self.path = []
       self.dx = [-1, 1, 0, 0]
       self.dy = [0, 0, -1, 1]
       self.rows = rows
       self.cols = cols
```

```
        self.world = world
        self.start = start
        self.goal = goal
        self.cost = 0
        self.covered = [[0 for x in range(0, cols)] for x in range(0, rows)]
        self.design_world()

        self.bfs()
        if self.reached == 0:
            print("No path exist.")

        print(self.path)
        print(self.cost)

        root.destroy()

    # get all the possible adjacent nodes of the given coordinates
    def get_adjacent(self, X, Y):
        adjacent = []
        for k in range(0, 4):
            x = X + self.dx[k]
            y = Y + self.dy[k]

            if(x >= 0 and x < self.rows and y >= 0 and y < self.cols and world[x][y] !=
2):
                adjacent.append([x, y])

        return adjacent

    def design_world(self):
        for i in range(rows):
            for j in range(cols):
                colorr = "black"
                if(world[i][j] == 2):
                    colorr = "red"
                elif world[i][j] == 3:
                    colorr = "yellow"
                canvas.create_rectangle(
                    i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill=colorr)

        root.update()
```

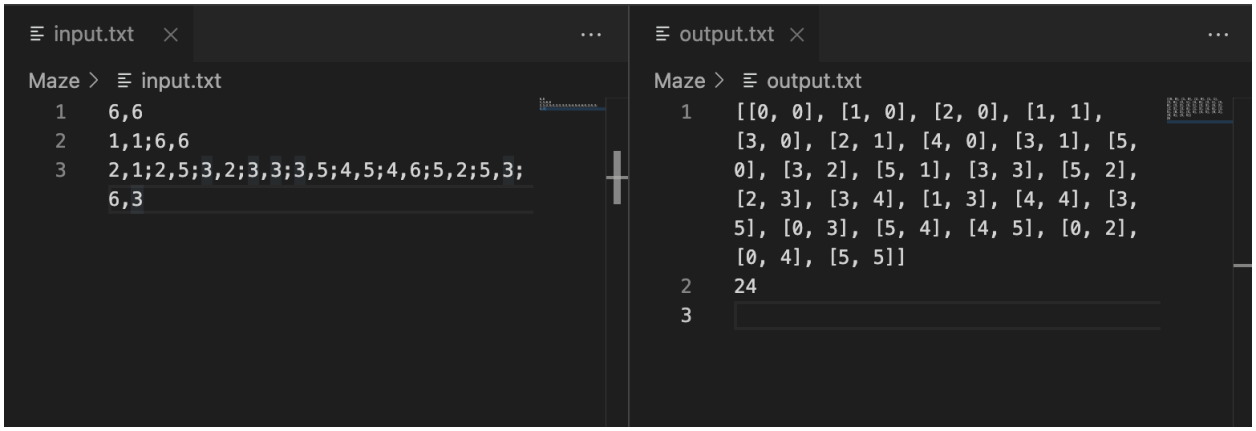
```
    def updateCanvas(self, cur):
        time.sleep(0.1)

        [i, j] = cur
        canvas.create_rectangle(
            i*blockSize + outlineLength, j*blockSize + outlineLength, (i+1)*blockSize,
(j+1)*blockSize, fill="#00B426", outline="#00B426")

        root.update()

    def bfs(self):
        queue = [self.start]

        while(len(queue)):
            top = queue[0]
            queue.pop(0)

            if self.reached == 1:
                break

            if self.covered[top[0]][top[1]] == 1:
                continue

            self.cost += 1

            self.covered[top[0]][top[1]] = 1
            self.path.append(top)
            self.updateCanvas(top)

            if top == self.goal:
                self.reached = 1
                break

            for [x, y] in self.get_adjacent(top[0], top[1]):
                queue.append([x, y])


if __name__ == "__main__":
    reached = 0

    rows, cols = map(int, input().split(","))
```

```
blockSize = min(windowWidth/rows, windowHeight/cols)
outlineLength = blockSize/15

xOffset = (windowWidth - blockSize*rows)/2
yOffset = (windowHeight - cols*blockSize)/2

start, end = input().split(";")
start = list(map(int, start.split(',')))
end = list(map(int, end.split(',')))

start[0] -= 1
start[1] -= 1
end[0] -= 1
end[1] -= 1

world = [[1 for x in range(0, cols)] for x in range(0, rows)]
world[0][0] = 0
world[end[0]][end[1]] = 3

obstacles = list(input().split(";"))
for i in obstacles:
    x, y = map(int, i.split(','))
    world[y-1][x-1] = 2

def runMaze():
    Maze(rows, cols, world, start, end)

root.after(1, runMaze)
root.mainloop()
```

BFS-UI Output (refer maze_bfs.gif in gifs folder)



Terminal Output:



——--

DLS:

Code:

```
from tkinter import *
import time

root = Tk()

windowWidth = 600
windowHeight = 600

blockSize = 0
outlineLength = 0

xOffset = 0
yOffset = 0

root.geometry(f'{windowWidth}x{windowHeight}')

canvas = Canvas(root, width=windowWidth, height=windowHeight, bg="black")
canvas.pack()



class Maze:

   def __init__(self, rows, cols, world, start, goal):
       self.reached = 0
       self.path = []
       self.dx = [-1, 1, 0, 0]
```

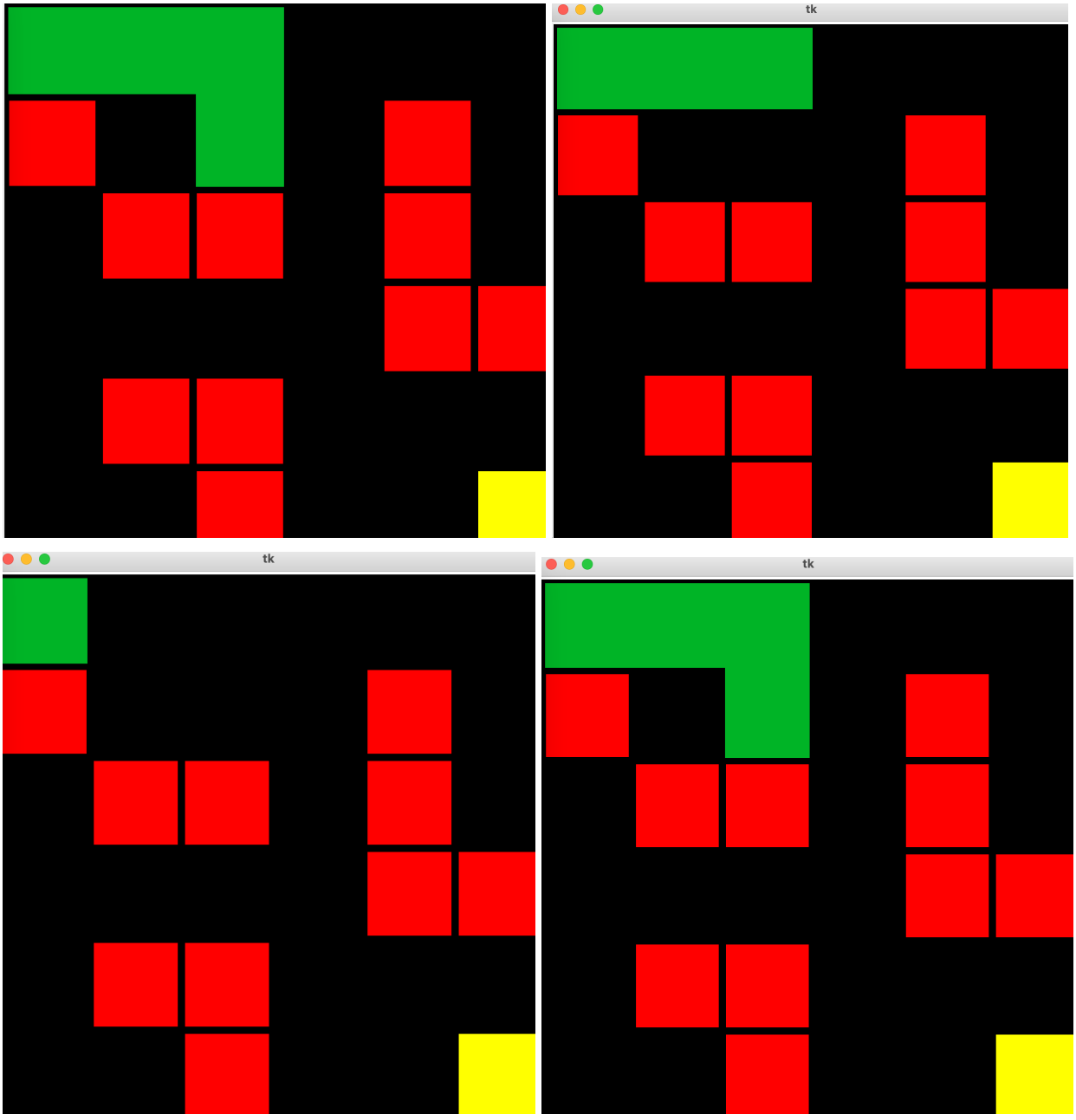
```
        self.dy = [0, 0, -1, 1]
        self.rows = rows
        self.cols = cols
        self.world = world
        self.start = start
        self.goal = goal
        self.covered = [[0 for x in range(0, cols)] for x in range(0, rows)]

        self.dfs(start, 0)
        print("DFS: ", self.path)
        if self.reached == 0:
            print("No path exist.")

        root.destroy()

    # get all the possible adjacent nodes of the given coordinates
    def get_adjacent(self, X, Y):
        adjacent = []
        for k in range(0, 4):
            x = X + self.dx[k]
            y = Y + self.dy[k]

            if(x >= 0 and x < self.rows and y >= 0 and y < self.cols):
                adjacent.append([x, y])

        return adjacent

    def updateCanvas(self):
        time.sleep(0.5)
        canvas.delete('all')

        for i in range(rows):
            for j in range(cols):
                canvas.create_rectangle(
                    i*blockSize, j*blockSize, (i+1)*blockSize, (j+1)*blockSize,
fill="black")

        for i in range(rows):
            for j in range(cols):
                colorr = "black"
                if(world[i][j] == 2):
                    colorr = "red"
```

```
                elif world[i][j] == 3:
                    colorr = "yellow"
                canvas.create_rectangle(
                    i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill=colorr)

        for [i, j] in self.path:
            canvas.create_rectangle(
                i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill="#00B426", outline="#00B426")

        for x in range(1, len(self.path)):
            i = (self.path[x][0] + self.path[x-1][0])/2
            j = (self.path[x][1] + self.path[x-1][1])/2
            canvas.create_rectangle(
                i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill="#00B426", outline="#00B426")

        root.update()

    def dls(self, cur, depth):

        if depth > 3:
            return

        # if already visited or have reached goal return
        if self.reached == 1 or self.covered[cur[0]][cur[1]] == 1:
            return

        # push the cur node in the path and mark this node visited
        self.path.append(cur)
        self.covered[cur[0]][cur[1]] = 1

        self.updateCanvas()

        # if current node is goal, mark reached as true
        if self.goal[0] == cur[0] and self.goal[1] == cur[1]:
            self.reached = 1
            return

        # traverse all the possible adjacent node of the current node
        for [x, y] in self.get_adjacent(cur[0], cur[1]):
```

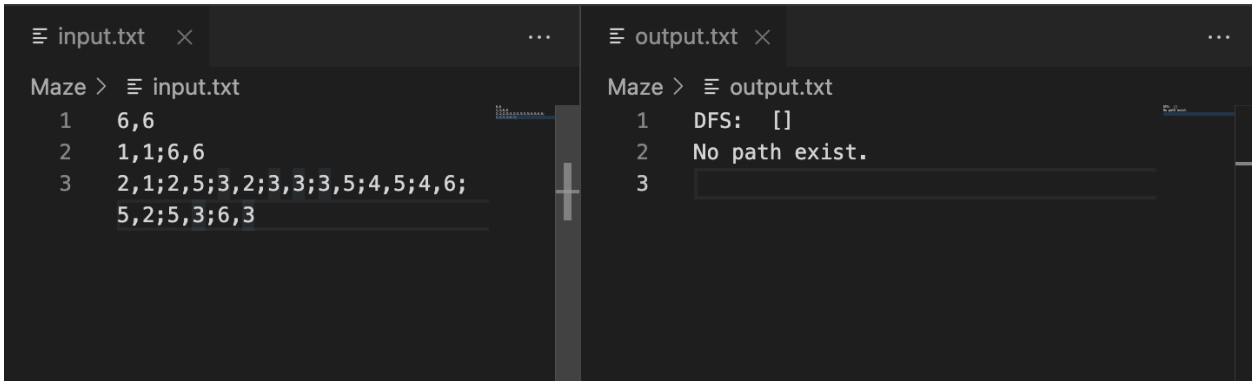
```
            if(self.world[x][y] != 2):
                self.dfs([x, y], depth + 1)

        # if cannot reach to the goal node through this node then pop this node
        if self.reached == 0:
            self.path.pop()

        self.updateCanvas()


if __name__ == "__main__":
    reached = 0

    rows, cols = map(int, input().split(","))

    blockSize = min(windowWidth/rows, windowHeight/cols)
    outlineLength = blockSize/15

    xOffset = (windowWidth - blockSize*rows)/2
    yOffset = (windowHeight - cols*blockSize)/2

    start, end = input().split(";")
    start = list(map(int, start.split(',')))
    end = list(map(int, end.split(',')))

    start[0] -= 1
    start[1] -= 1
    end[0] -= 1
    end[1] -= 1

    world = [[1 for x in range(0, cols)] for x in range(0, rows)]
    world[0][0] = 0
    world[end[0]][end[1]] = 3

    obstacles = list(input().split(";"))
    for i in obstacles:
        x, y = map(int, i.split(','))
        world[y-1][x-1] = 2

    def runMaze():
        Maze(rows, cols, world, start, end)
```

```
    root.after(1, runMaze)
    root.mainloop()
```

DLS-UI Output (refer maze_dls.gif in gifs folder)



Terminal Output:



Explanation:

In DLS, as we are given only the height of three so we only need to do dfs up to height of given if node found then well and good otherwise, we will terminate the dfs.

—---

UCS

Code:

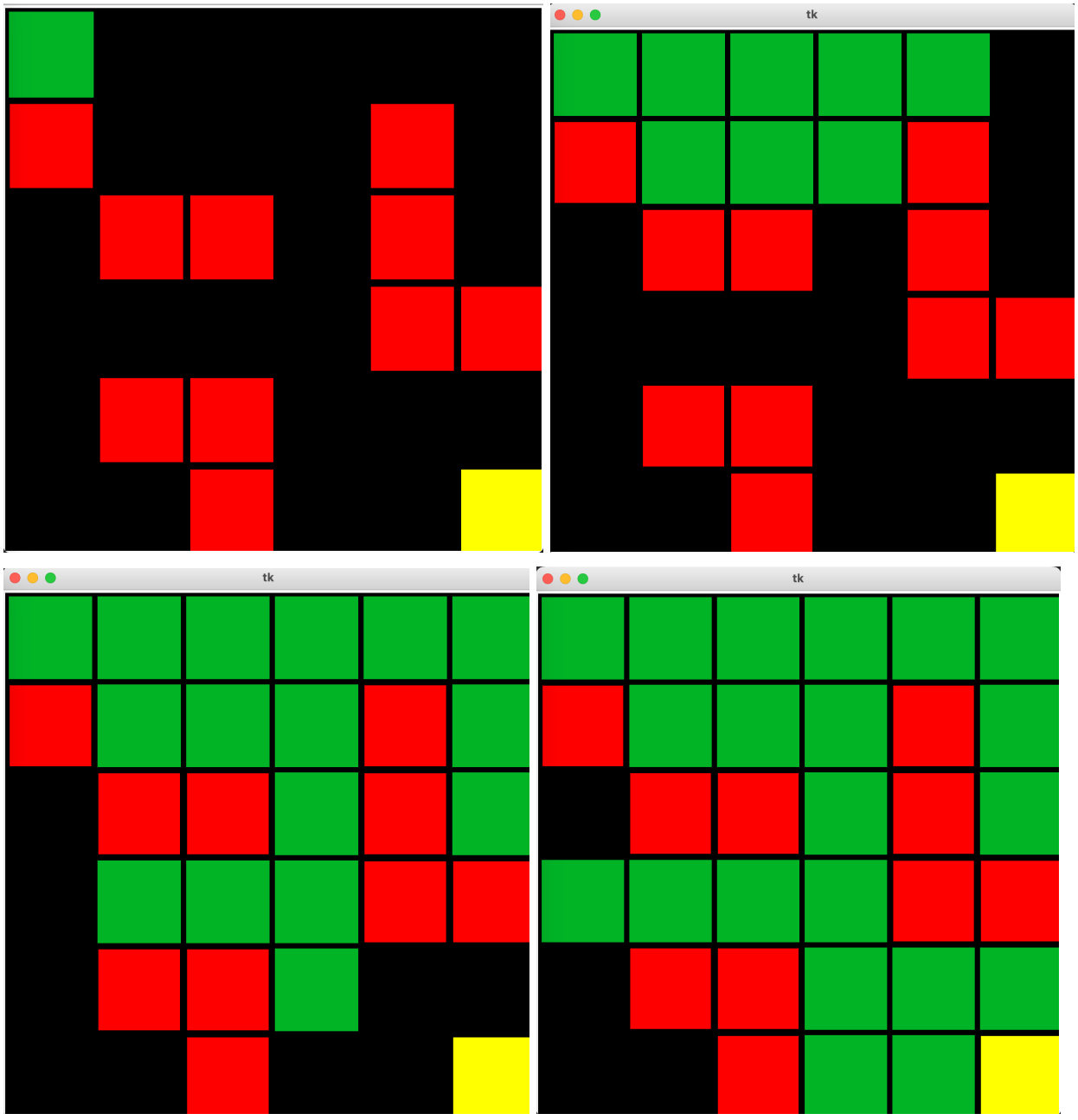
```
from tkinter import *
import time
from queue import PriorityQueue

root = Tk()

windowWidth = 600
windowHeight = 600

blockSize = 0
outlineLength = 0

xOffset = 0
yOffset = 0

root.geometry(f'{windowWidth}x{windowHeight}')

canvas = Canvas(root, width=windowWidth, height=windowHeight, bg="black")
canvas.pack()


class Maze:
```

```
    def __init__(self, rows, cols, world, start, goal):
        self.reached = 0
        self.path = []
        self.dx = [-1, 1, 0, 0]
        self.dy = [0, 0, -1, 1]
        self.rows = rows
        self.cost = 0
        self.cols = cols
        self.world = world
        self.start = start
        self.goal = goal
        self.covered = [[0 for x in range(0, cols)] for x in range(0, rows)]
        self.design_world()

        self.ucs()
        if self.reached == 0:
            print("No path exist.")
        print(self.path)
        print("Cost: ", self.cost)

        root.destroy()

    # get all the possible adjacent nodes of the given coordinates
    def get_adjacent(self, X, Y):
        adjacent = []
        for k in range(0, 4):
            x = X + self.dx[k]
            y = Y + self.dy[k]

            if(x >= 0 and x < self.rows and y >= 0 and y < self.cols and world[x][y] !=
2):
                adjacent.append([x, y])

        return adjacent

    def design_world(self):
        for i in range(rows):
            for j in range(cols):
                colorr = "black"
                if(world[i][j] == 2):
                    colorr = "red"
                elif world[i][j] == 3:
```

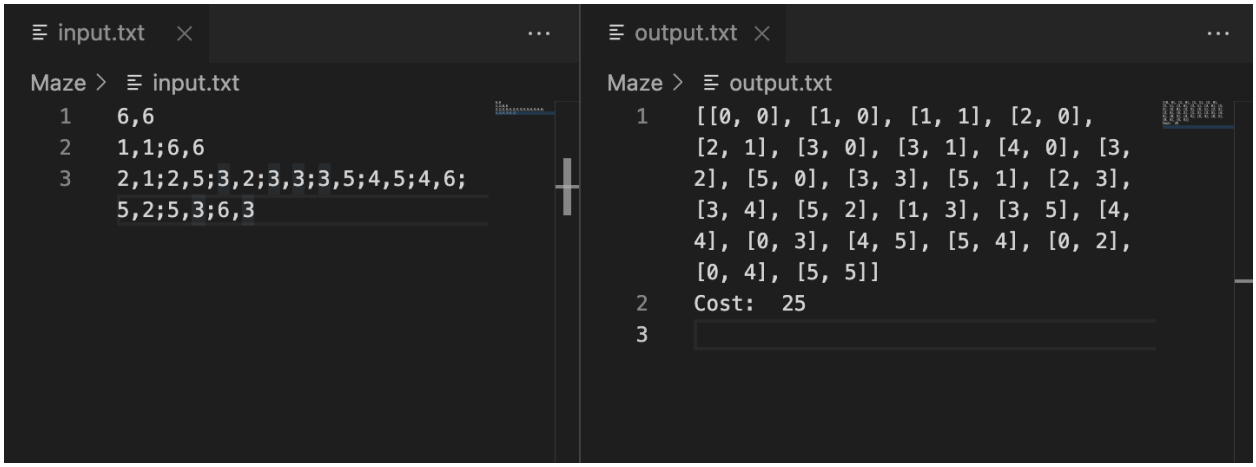
```
                    colorr = "yellow"
                canvas.create_rectangle(
                    i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill=colorr)

        root.update()

    def updateCanvas(self, cur):
        time.sleep(0.1)

        [i, j] = cur
        canvas.create_rectangle(
            i*blockSize + outlineLength, j*blockSize + outlineLength, (i+1)*blockSize,
(j+1)*blockSize, fill="#00B426", outline="#00B426")

        root.update()

    def ucs(self):

        que = PriorityQueue()
        [X, Y] = self.start
        que.put([0, self.start])

        while(que.empty() == False):

            top = que.get()

            if self.covered[top[1][0]][top[1][1]] == 1:
                continue

            self.covered[top[1][0]][top[1][1]] = 1
            self.cost += 1
            if self.reached == 1:
                break

            self.path.append(top[1])
            self.updateCanvas(top[1])

            if top[1] == self.goal:
                self.reached = 1

            for [x, y] in self.get_adjacent(top[1][0], top[1][1]):
```

```
                fx = top[0] + 1
                que.put([fx, [x, y]])


if __name__ == "__main__":
    reached = 0

    rows, cols = map(int, input().split(","))

    blockSize = min(windowWidth/rows, windowHeight/cols)
    outlineLength = blockSize/15

    xOffset = (windowWidth - blockSize*rows)/2
    yOffset = (windowHeight - cols*blockSize)/2

    start, end = input().split(";")
    start = list(map(int, start.split(',')))
    end = list(map(int, end.split(',')))

    start[0] -= 1
    start[1] -= 1
    end[0] -= 1
    end[1] -= 1

    world = [[1 for x in range(0, cols)] for x in range(0, rows)]
    world[0][0] = 0
    world[end[0]][end[1]] = 3

    obstacles = list(input().split(";"))
    for i in obstacles:
        x, y = map(int, i.split(','))
        world[y-1][x-1] = 2

    def runMaze():
        Maze(rows, cols, world, start, end)

    root.after(1, runMaze)
    root.mainloop()
```

UCS-UI Output (refer maze_ucs.gif in gifs folder)



Terminal Output:



—---

GBFS:

Code:

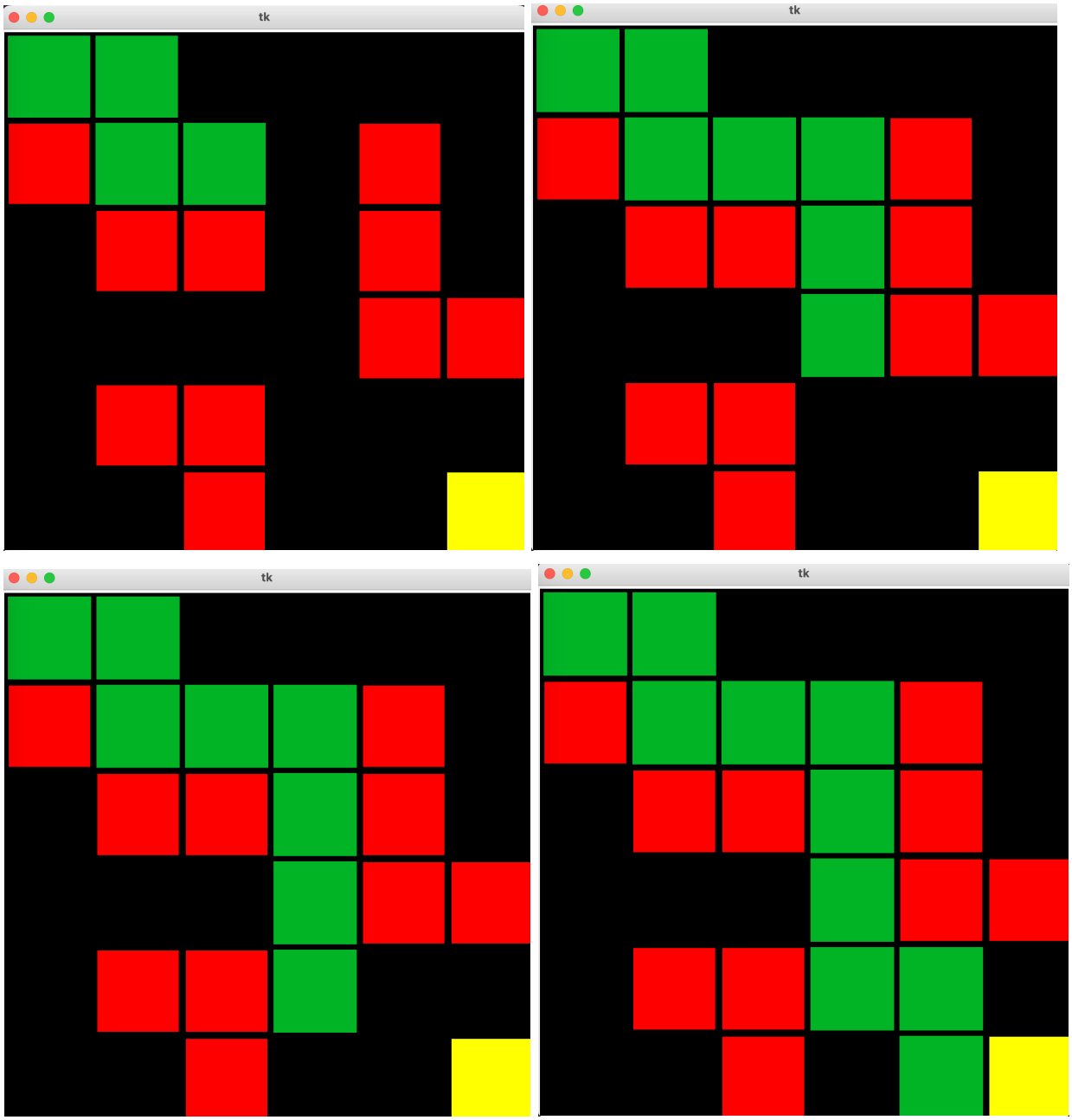
```
from tkinter import *
import time
from queue import PriorityQueue

root = Tk()

windowWidth = 600
windowHeight = 600

blockSize = 0
outlineLength = 0

xOffset = 0
yOffset = 0

root.geometry(f'{windowWidth}x{windowHeight}')

canvas = Canvas(root, width=windowWidth, height=windowHeight, bg="black")
canvas.pack()


class Maze:

   def __init__(self, rows, cols, world, start, goal):
       self.reached = 0
```

```
        self.path = []
        self.dx = [-1, 1, 0, 0]
        self.dy = [0, 0, -1, 1]
        self.rows = rows
        self.cost = 0
        self.cols = cols
        self.world = world
        self.start = start
        self.goal = goal
        self.covered = [[0 for x in range(0, cols)] for x in range(0, rows)]
        self.design_world()

        self.gbfs()
        if self.reached == 0:
            print("No path exist.")
        print(self.path)
        print("Cost: ", self.cost)

        root.destroy()

    # get all the possible adjacent nodes of the given coordinates
    def get_adjacent(self, X, Y):
        adjacent = []
        for k in range(0, 4):
            x = X + self.dx[k]
            y = Y + self.dy[k]

            if(x >= 0 and x < self.rows and y >= 0 and y < self.cols and world[x][y] !=
2):
                adjacent.append([x, y])

        return adjacent

    def design_world(self):
        for i in range(rows):
            for j in range(cols):
                colorr = "black"
                if(world[i][j] == 2):
                    colorr = "red"
                elif world[i][j] == 3:
                    colorr = "yellow"
                canvas.create_rectangle(
```

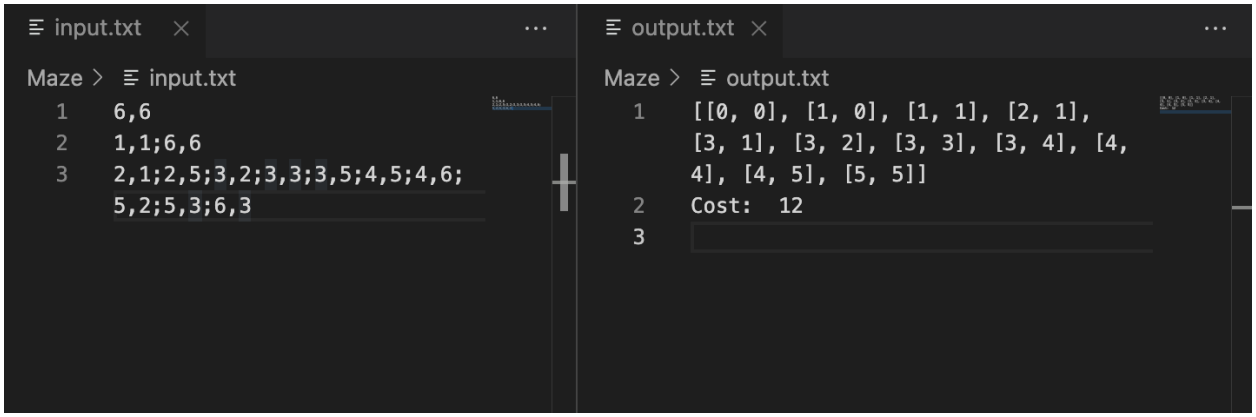
```
                    i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill=colorr)

        root.update()

    def updateCanvas(self, cur):
        time.sleep(0.1)

        [i, j] = cur
        canvas.create_rectangle(
            i*blockSize + outlineLength, j*blockSize + outlineLength, (i+1)*blockSize,
(j+1)*blockSize, fill="#00B426", outline="#00B426")

        root.update()

    def gbfs(self):

        que = PriorityQueue()
        [x, y] = self.start
        [X, Y] = self.goal
        hx = pow(abs(x-X), 2) + pow(abs(y-Y), 2)
        que.put([hx, self.start])

        while(que.empty() == False):

            top = que.get()

            if self.covered[top[1][0]][top[1][1]] == 1:
                continue

            self.cost += 1
            self.covered[top[1][0]][top[1][1]] = 1

            if self.reached == 1:
                break

            self.path.append(top[1])
            self.updateCanvas(top[1])

            if top[1] == self.goal:
                self.reached = 1
```

```
            for [x, y] in self.get_adjacent(top[1][0], top[1][1]):
                hx = pow(abs(x-X), 2) + pow(abs(y-Y), 2)
                que.put([hx, [x, y]])


if __name__ == "__main__":
    reached = 0

    rows, cols = map(int, input().split(","))

    blockSize = min(windowWidth/rows, windowHeight/cols)
    outlineLength = blockSize/15

    xOffset = (windowWidth - blockSize*rows)/2
    yOffset = (windowHeight - cols*blockSize)/2

    start, end = input().split(";")
    start = list(map(int, start.split(',')))
    end = list(map(int, end.split(',')))

    start[0] -= 1
    start[1] -= 1
    end[0] -= 1
    end[1] -= 1

    world = [[1 for x in range(0, cols)] for x in range(0, rows)]
    world[0][0] = 0
    world[end[0]][end[1]] = 3

    obstacles = list(input().split(";"))
    for i in obstacles:
        x, y = map(int, i.split(','))
        world[y-1][x-1] = 2

    def runMaze():
        Maze(rows, cols, world, start, end)

    root.after(1, runMaze)
    root.mainloop()
```

GBFS-UI Output (refer maze_gbfs.gif in gifs folder) :



Terminal Output:



—---

Astar:

Code:

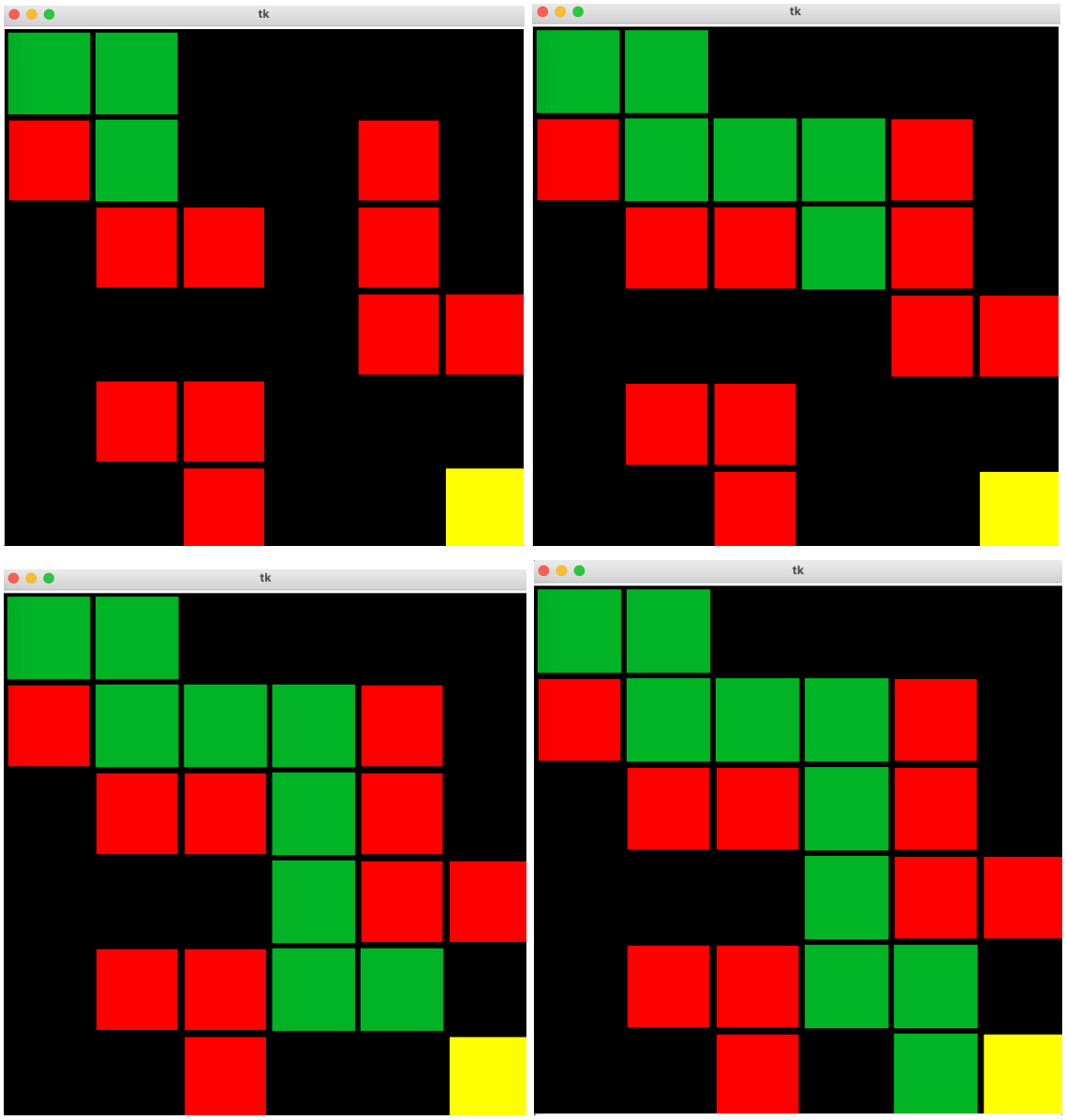
```
from tkinter import *
import time
from queue import PriorityQueue

root = Tk()

windowWidth = 600
windowHeight = 600

blockSize = 0
outlineLength = 0

xOffset = 0
yOffset = 0

root.geometry(f'{windowWidth}x{windowHeight}')

canvas = Canvas(root, width=windowWidth, height=windowHeight, bg="black")
canvas.pack()



class Maze:

   def __init__(self, rows, cols, world, start, goal):
       self.reached = 0
       self.path = []
       self.dx = [-1, 1, 0, 0]
       self.dy = [0, 0, -1, 1]
```

```
        self.rows = rows
        self.cost = 0
        self.cols = cols
        self.world = world
        self.start = start
        self.goal = goal
        self.covered = [[0 for x in range(0, cols)] for x in range(0, rows)]
        self.design_world()

        self.astar()
        if self.reached == 0:
            print("No path exist.")
        print(self.path)
        print("Cost: ", self.cost)

        root.destroy()

    # get all the possible adjacent nodes of the given coordinates
    def get_adjacent(self, X, Y):
        adjacent = []
        for k in range(0, 4):
            x = X + self.dx[k]
            y = Y + self.dy[k]

            if(x >= 0 and x < self.rows and y >= 0 and y < self.cols and world[x][y] !=
2):
                adjacent.append([x, y])

        return adjacent

    def design_world(self):
        for i in range(rows):
            for j in range(cols):
                colorr = "black"
                if(world[i][j] == 2):
                    colorr = "red"
                elif world[i][j] == 3:
                    colorr = "yellow"
                canvas.create_rectangle(
                    i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill=colorr)
```

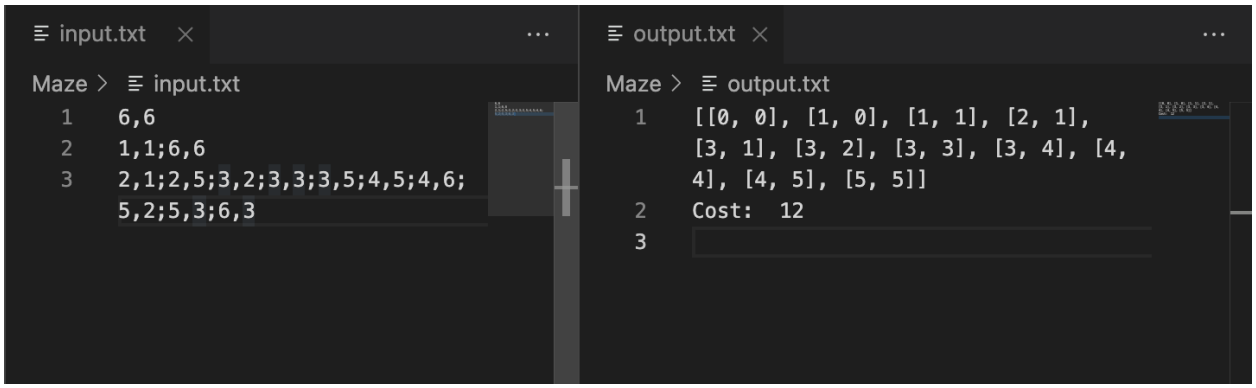
```
        root.update()

    def updateCanvas(self, cur):
        time.sleep(0.1)

        [i, j] = cur
        canvas.create_rectangle(
            i*blockSize + outlineLength, j*blockSize + outlineLength, (i+1)*blockSize,
(j+1)*blockSize, fill="#00B426", outline="#00B426")

        root.update()

    def astar(self):

        que = PriorityQueue()
        [x, y] = self.start
        [X, Y] = self.goal
        hx = pow(abs(x-X), 2) + pow(abs(y-Y), 2)
        que.put([0 + hx, 0, hx, self.start])

        while(que.empty() == False):

            top = que.get()

            if self.covered[top[3][0]][top[3][1]] == 1:
                continue

            self.cost += 1
            self.covered[top[3][0]][top[3][1]] = 1

            if self.reached == 1:
                break

            self.path.append(top[3])
            self.updateCanvas(top[3])

            if top[3] == self.goal:
                self.reached = 1

            for [x, y] in self.get_adjacent(top[3][0], top[3][1]):
                hx = pow(abs(x-X), 2) + pow(abs(y-Y), 2)
                fx = top[1]
```

```
                que.put([hx + fx, fx, hx, [x, y]])


if __name__ == "__main__":
    reached = 0

    rows, cols = map(int, input().split(","))

    blockSize = min(windowWidth/rows, windowHeight/cols)
    outlineLength = blockSize/15

    xOffset = (windowWidth - blockSize*rows)/2
    yOffset = (windowHeight - cols*blockSize)/2

    start, end = input().split(";")
    start = list(map(int, start.split(',')))
    end = list(map(int, end.split(',')))

    start[0] -= 1
    start[1] -= 1
    end[0] -= 1
    end[1] -= 1

    world = [[1 for x in range(0, cols)] for x in range(0, rows)]
    world[0][0] = 0
    world[end[0]][end[1]] = 3

    obstacles = list(input().split(";"))
    for i in obstacles:
        x, y = map(int, i.split(','))
        world[y-1][x-1] = 2

    def runMaze():
        Maze(rows, cols, world, start, end)

    root.after(1, runMaze)
    root.mainloop()
```

Astar-UI Output (refer maze_astar.gif in gifs folder) :



Terminal Output:



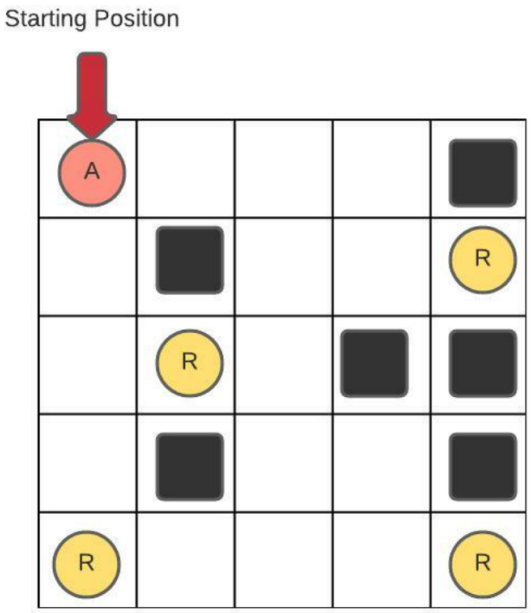
Observation:

- I have used tkinter to show visualization of all algorithms separately.
- I have used manhattan and euclidean distance to get the heuristic function
- In **DFS**, we have to first completely explore the one child in depth then the second or other childs are explored in the same manner.
- **Breadth-first search** (**BFS**) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.
- **Depth limited search** is an uninformed search algorithm which is similar to Depth First Search(DFS). It can be considered equivalent to DFS with a predetermined depth limit 'l'. Nodes at depth l are considered to be nodes without any successors.
- **Uniform-cost search** is an uninformed search algorithm that uses the lowest cumulative cost to find a path from the source to the destination. Nodes are expanded, starting from the root, according to the minimum cumulative cost.
- **Greedy best-first search** algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search.
- **A* Star** a searching algorithm that is used to find the shortest path between an initial and a final point.

—---

Maze Problem with multi-goal [10 Marks]

Consider the maze given in the figure below. The walled tiles are marked in black and your agent A cannot move to or through those positions.



Inputs:
Write python/C program that takes the maze as a 5x5 matrix input where 0 denotes an empty tile, 1 denotes an obstruction/wall, 2 denotes the start state and 3 denotes the reward. Assume valid actions as L,R,U,D,S,N where L=move_left, R=move_right, U=move_up, D=move_down.

Use the **A* algorithms as (astar)** on the resultant maze for your agent to reach all the rewards, and keep a record of the tiles visited on the way.

Code:

```
from tkinter import *
import time
import math
from queue import PriorityQueue

root = Tk()

windowWidth = 600
windowHeight = 600

blockSize = 0
outlineLength = 0
```

```
xOffset = 0
yOffset = 0

root.geometry(f'{windowWidth}x{windowHeight}')

canvas = Canvas(root, width=windowWidth, height=windowHeight, bg="black")
canvas.pack()


class ASTAR:

   def __init__(self, rows, cols, world, start, goal):
       self.reached = 0
       self.path = []
       self.dx = [-1, 1, 0, 0]
       self.dy = [0, 0, -1, 1]
       self.rows = rows
       self.cost = 0
       self.cols = cols
       self.world = world
       self.start = start
       self.goal = goal
       self.covered = [[0 for x in range(0, cols)] for x in range(0, rows)]

       self.astar()

   def getResult(self):
       return (self.path, self.cost)

   # get all the possible adjacent nodes of the given coordinates
   def get_adjacent(self, X, Y):
       adjacent = []
       for k in range(0, 4):
           x = X + self.dx[k]
           y = Y + self.dy[k]

           if(x >= 0 and x < self.rows and y >= 0 and y < self.cols and world[x][y] !=
2):
               adjacent.append([x, y])

       return adjacent
```

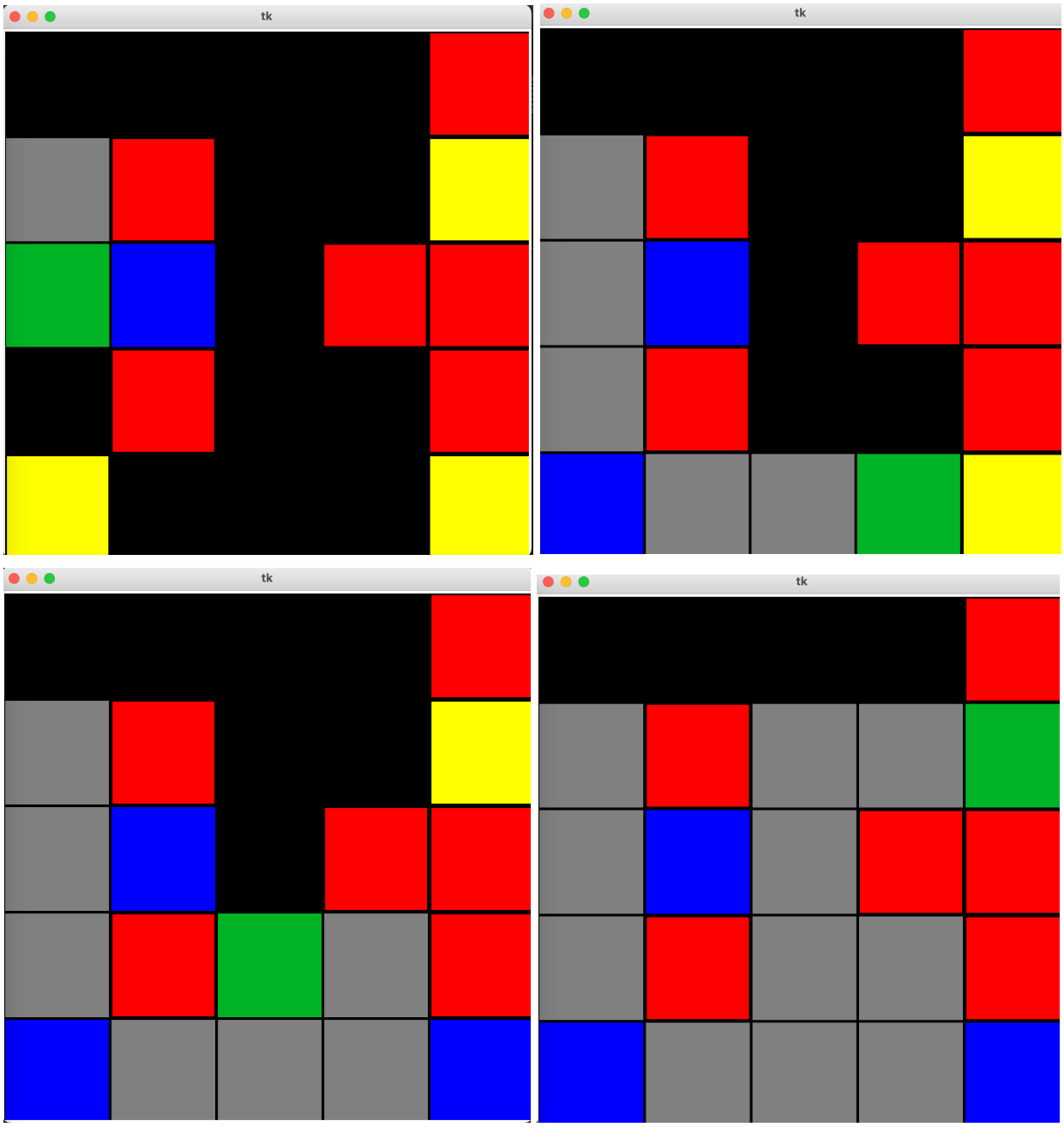
```
  def astar(self):

      que = PriorityQueue()
      [x, y] = self.start
      [X, Y] = self.goal
      hx = pow(abs(x-X), 2) + pow(abs(y-Y), 2)
      que.put([0 + hx, 0, hx, self.start, [self.start]])

      while(que.empty() == False):

          top = que.get()
          path = top[4]

          if self.covered[top[3][0]][top[3][1]] == 1:
              continue

          self.cost += 1
          self.covered[top[3][0]][top[3][1]] = 1

          if top[3] == self.goal:
              self.path = path
              return

          for [x, y] in self.get_adjacent(top[3][0], top[3][1]):
              hx = pow(abs(x-X), 2) + pow(abs(y-Y), 2)
              fx = top[1]
              newPath = path + [[x, y]]
              que.put([hx + fx, fx, hx, [x, y], newPath])


class Maze:

  def __init__(self, rows, cols, world, start, goals):
      self.path = []
      self.dx = [-1, 1, 0, 0]
      self.dy = [0, 0, -1, 1]
      self.rows = rows
      self.cols = cols
      self.world = world
      self.start = start
      self.goals = goals
```

```
        self.cost = 0

        self.multigoalAstar()

        print(self.path)
        print(self.cost)

        self.updateCanvas()

        root.destroy()

    def updateCanvas(self):
        canvas.delete('all')

        for i in range(rows):
            for j in range(cols):
                canvas.create_rectangle(
                    i*blockSize, j*blockSize, (i+1)*blockSize, (j+1)*blockSize,
fill="black")

        for i in range(rows):
            for j in range(cols):
                colorr = "black"
                if(world[i][j] == 2):
                    colorr = "red"
                elif world[i][j] == 3:
                    colorr = "yellow"
                canvas.create_rectangle(
                    i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill=colorr)

        root.update()

        for [i, j] in self.path:

            canvas.create_rectangle(
                i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill="#00B426", outline="#00B426")

            root.update()
            time.sleep(0.4)
```

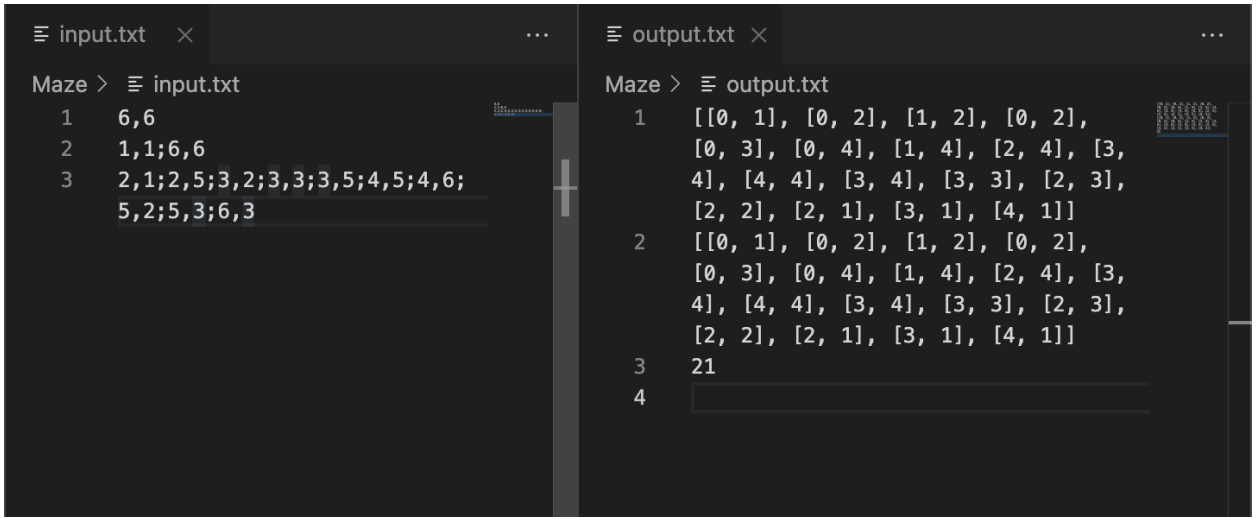
```
            colorr = "grey"
            if(world[i][j] == 3):
                colorr = "blue"

            canvas.create_rectangle(
                i*blockSize + outlineLength, j*blockSize + outlineLength,
(i+1)*blockSize, (j+1)*blockSize, fill=colorr, outline=colorr)

            root.update()

    def getManhattan(self, start, goal):
        return abs(start[0] - goal[0]) + abs(start[1] - goal[1])

    def multigoalAstar(self):
        cnt = 0
        vis = []
        start = self.start
        while cnt < len(self.goals):
            goal = []
            cost = math.inf

            for end in self.goals:
                if end not in vis and cost > self.getManhattan(start, end):
                    cost = self.getManhattan(start, end)
                    goal = end

            obj = ASTAR(self.rows, self.cols, self.world, start, goal)
            (path, cost) = obj.getResult()
            self.path += path[1:]
            start = goal
            self.cost += cost
            vis.append(goal)
            cnt += 1

        print(self.path)


if __name__ == "__main__":
    reached = 0

    rows, cols = map(int, input().split(","))
```

```
blockSize = min(windowWidth/rows, windowHeight/cols)
outlineLength = blockSize/30

xOffset = (windowWidth - blockSize*rows)/2
yOffset = (windowHeight - cols*blockSize)/2

start = list(map(int, input().split(',')))
start[0] -= 1
start[1] -= 1

end = list(input().split(";"))
for j in range(len(end)):
    end[j] = list(map(int, end[j].split(',')))
    end[j][0] -= 1
    end[j][1] -= 1

world = [[1 for x in range(0, cols)] for x in range(0, rows)]
world[0][0] = 0
for [x, y] in end:
    world[x][y] = 3

obstacles = list(input().split(";"))
for i in obstacles:
    x, y = map(int, i.split(','))
    world[x-1][y-1] = 2

def runMaze():
    Maze(rows, cols, world, start, end)

root.after(1, runMaze)
root.mainloop()
```

Astar-MultiGoal-UI Output (refer maze_astar_multigoal_ui.gif in gifs folder) :



Terminal Output:



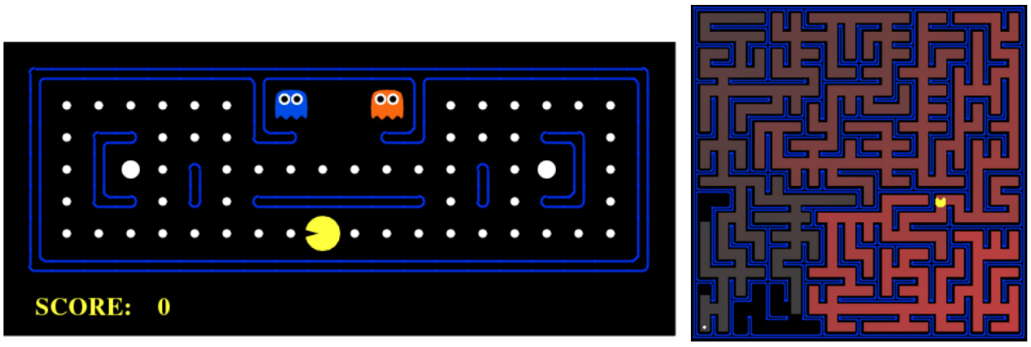
Observation:

1. Here we are using A* for multiple points. Initially we use initial start state as start state and with A* we will find the first goal which will be least in cost + heuristic and it will be reached and then this state will be use as start state and other non-visited goal state are goal state, we again use recursively A* Algo to move to next goal state, in this way we will reach to all goal state.
2. We have use the tkinter to visualize the movement of the agent.

PART A : Exploratory Problem [25 Marks]

4. **Search in Pac-Man** This problem allow you to visualize the results of the techniques you implement. Pac-Man provides a challenging problem environment that demands creative solutions of a real-world AI problems. The Pacman agent needs to find paths through the maze world, both to reach a location and to collect food efficiently. In this Problem, you are expected to implement and experiment with different AI search techniques that was discussed in the class in a Pacman environment.

   This lab assignment is inspired by Project 1: Search, which is a part of a recent offering of CS188 at UC Berkeley[1]. We thank the authors at Berkeley for making their project available to the public.



Aim: Students implement depth-first, breadth-first, uniform cost, and A* search algorithms. These algorithms are used to solve navigation and traveling salesman problems in the Pacman world.

The code for this assignment is provided to you as LA_4_5_search zip folder. You can download all the code and supporting files as a Folder/ zip archive from the shared link. The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

Assumption : The projects for this class assume you use Python 3.6.

Files you'll edit:

`search.py` :	Where all of your search algorithms will reside.
`searchAgents.py` :	Where all of your search-based agents will reside.

Question 1 (3 points) : Finding a Fixed Food Dot using Depth First Search

In `searchAgents.py`, you'll find a fully implemented `SearchAgent`, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented – that's your job.

First, test that the `SearchAgent` is working correctly by running:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

The command above tells the `SearchAgent` to use `tinyMazeSearch` as its search algorithm, which is implemented in `search.py`. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Your code should quickly find a solution for:

```
python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

python pacman.py -l bigMaze -z .5 -p SearchAgent
```

Code:

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """

    from util import Stack

    st = Stack()
    path = []
    vis = []

    st.push((problem.getStartState(), []))

    while(True):
        if st.isEmpty():
            return []

        cur, path = st.pop()
        if cur in vis:
            continue

        vis.append(cur)
```

```
    if problem.isGoalState(cur):
        return path

    successor = problem.getSuccessors((cur))

    for succ in successor:
        if succ[0] not in vis:
            st.push((succ[0], path + [succ[1]]))

util.raiseNotDefined()
```

Output:

1. python pacman.py -l tinyMaze -p SearchAgent

```
[sh-3.2$ python3 pacman.py -l tinyMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$
```

2. python pacman.py -l mediumMaze -p SearchAgent

```
[sh-3.2$ python3 pacman.py -l mediumMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$ _
```

3. python pacman.py -l mediumMaze -p SearchAgent

```
sh-3.2$ python3 pacman.py -l bigMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$
```

Observation:

In this problem we can complete DFS function in search.py file and run successfully on all type the maze. And in DFS we explored the one of the first node of each node and then one of its adjacent first node. And by using the internal Stack implemented in Utill.py file and keep storing its path.

Question 2 (3 points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `search.py`. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

Does BFS find a least cost solution? If not, check your implementation.

Hint: If Pacman moves too slowly for you, try the option `--frameTime 0`.

Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

Code:

```
def breadthFirstSearch(problem):
   """Search the shallowest nodes in the search tree first."""

   startState = problem.getStartState()
   queue = [(startState, [])]
   vis = []

   while(len(queue)):
       top, path = queue[0]
```

```
        queue.pop(0)

        if top in vis:
            continue

        vis.append(top)
        if problem.isGoalState((top)):
            return path

        successors = problem.getSuccessors((top))
        for succ in successors:
            print(succ)
            if succ[0] not in vis:
                queue.append((succ[0], path + [succ[1]]))

    return []

    util.raiseNotDefined()
```

Output:

1. python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

```
sh-3.2$ python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$ _
```

2. python pacman.py -l bigMaze -p SearchAgent -a fn=bfs

```
sh-3.2$ python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$
```

Observation:

In this problem we can implement the BFS. We have used the internal Queue implemented in Utill.py file to perform BFS. And we will keep storing individual paths.

Question 3 (3 points): Varying the Cost Function

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider `mediumDottedMaze` and `mediumScaryMaze`.

By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in `search.py`. We encourage you to look through `util.py` for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Code:

```
def uniformCostSearch(problem):
   """Search the node of least total cost first."""

   from util import PriorityQueue

   que = PriorityQueue()
   vis = []
   path = []

   start = problem.getStartState()
   que.push((start, []), 0)

   while(True):
       if que.isEmpty():
           return []
```

```
        top, path = que.pop()

        if top in vis:
            continue
        vis.append(top)

        if problem.isGoalState((top)):
            return path

        successors = problem.getSuccessors((top))
        for succ in successors:
            if succ[0] not in vis:
                newPath = path + [succ[1]]
                cost = problem.getCostOfActions(newPath)
                que.push((succ[0], newPath), cost)

    return []

    util.raiseNotDefined()
```

Output:

1. python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

```
sh-3.2$ python3 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$ _
```

2. python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

```
sh-3.2$ python3 pacman.py -l mediumDottedMaze -p StayEastSearchAgent
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$ _
```

3. python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

```
sh-3.2$ python3 pacman.py -l mediumScaryMaze -p StayWestSearchAgent
Path found with total cost of 68719479864 in 0.0 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:        418.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$
```

Observation:

In this problem we have implemented the uniform cost function in the search.py file. And successfully able to complete both medium maze and medium dotted maze. Here we have used the priority queue which is internally implemented in the Util.py file.

Question 4 (3 points): A* search

Implement A* graph search in the empty function **aStarSearch** in **search.py**. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The **nullHeuristic** heuristic function in **search.py** is a trivial example.

You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as **manhattanHeuristic** in **searchAgents.py**).

Code:

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""

    from util import PriorityQueue

    que = PriorityQueue()
    vis = []
    path = []

    start = problem.getStartState()
    hx = heuristic(start, problem)
    que.push((start, []), hx)
```

```
    while(True):
        if que.isEmpty():
            return []

        top, path = que.pop()

        if top in vis:
            continue
        vis.append(top)

        if problem.isGoalState((top)):
            return path

        successors = problem.getSuccessors((top))
        for succ in successors:
            if succ[0] not in vis:
                newPath = path + [succ[1]]
                fx = problem.getCostOfActions(newPath)
                hx = heuristic(succ[0], problem)
                que.push((succ[0], newPath), fx + hx)

    return []

    util.raiseNotDefined()
```

Output:

1. python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar, heuristic=manhattanHeuristic

```
[sh-3.2$ python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=]
manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$
```

Observation:

The same function can also work for different mazes. As in above ss A* is implemented in openMaze. We will get solution for all those mazes also with same method of A*.

Question 5 (3 points): Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

In *corner mazes*, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like `tinyCorners`, the shortest path does not always go to the closest food first! *Hint*: the shortest path through `tinyCorners` takes 28 steps.

Note: Make sure to complete Question 2 before working on Question 5, because Question 5 builds upon your answer for Question 2.

Implement the `CornersProblem` search problem in `searchAgents.py`. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, your search agent should solve:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

```
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

To receive full credit, you need to define an abstract state representation that *does not* encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman `GameState` as a search state. Your code will be very, very slow if you do (and also wrong).

Hint: The only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners.

Our implementation of `breadthFirstSearch` expands just under 2000 search nodes on `mediumCorners`. However, heuristics (used with A* search) can reduce the amount of searching required.

Output:

1. python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

```
sh-3.2$ python3 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersPr
oblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$ _
```

2. python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

```
sh-3.2$ python3 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=Corners
Problem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.2 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$ _
```

Observation:

In this we need to implement some of the function in the CornersProblem class line getStartState, getInitialState,GetGoalState etc. and after that we are successfully able to run the packman in different mazes.

Question 6 (3 points): Corners Problem: Heuristic

Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.

Implement a non-trivial, consistent heuristic for the `CornersProblem` in `cornersHeuristic`.

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

Note: `AStarCornersAgent` is a shortcut for

```
-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

Output:

1. python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

```
sh-3.2$ python3 pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 1136
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$ _
```

2. python3 pacman.py -l mediumCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic

```
sh-3.2$ python3 pacman.py -l mediumCorners -p SearchAgent -a fn=aStarSearch,prob
=CornersProblem,heuristic=cornersHeuristic
[SearchAgent] using function aStarSearch and heuristic cornersHeuristic
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 1136
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$
```

Observations:

We explore near to 1136 nodes and can eat all corner points by implementing all required functions in searchAgent.py file.

In this problem we need to implement the cornerHeuristic function in searchAgent.py file and according to that heuristic only successor selection happens in cornerProblem class.

Question 7 (4 points): Eating All The Dots

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem: `FoodSearchProblem` in `searchAgents.py` (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) If you have written your general search methods correctly, `A*` with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to `testSearch` with no code change on your part (total cost of 7).

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

Note: `AStarFoodSearchAgent` is a shortcut for

```
-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
```

You should find that UCS starts to slow down even for the seemingly simple `tinySearch`. As a reference, our implementation takes 2.5 seconds to find a path of length 27 after expanding 5057 search nodes.

Code:

1. python pacman.py -l testSearch -p AStarFoodSearchAgent

```
sh-3.2$ python3 pacman.py -l testSearch -p AStarFoodSearchAgent
Path found with total cost of 7 in 0.0 seconds
Search nodes expanded: 10
Pacman emerges victorious! Score: 513
Average Score: 513.0
Scores:        513.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$
```

2. python3 pacman.py -l testSearch -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic

```
sh-3.2$ python3 pacman.py -l testSearch -p SearchAgent -a fn=astar,prob=FoodSear
chProblem,heuristic=foodHeuristic
[SearchAgent] using function astar and heuristic foodHeuristic
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 7 in 0.0 seconds
Search nodes expanded: 10
Pacman emerges victorious! Score: 513
Average Score: 513.0
Scores:        513.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$
```

Observation:

In this problem we have implemented the heuristic function of FoodSearchProblem, as we are given the positions of foods as a list. And our heuristic is exploring 4299 nodes to get all the foods. We are able to run all the given commands properly.

Question 8 (3 points): Suboptimal Search

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we'd still like to find a reasonably good path, quickly. In this section, you'll write an agent that always greedily eats the closest dot. `ClosestDotSearchAgent` is implemented for you in `searchAgents.py`, but it's missing a key function that finds a path to the closest dot.

Implement the function `findPathToClosestDot` in `searchAgents.py`. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350:

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Output:

1. python3 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

```
sh-3.2$ python3 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with cost 350.
Pacman emerges victorious! Score: 2360
Average Score: 2360.0
Scores:        2360.0
Win Rate:      1/1 (1.00)
Record:        Win
sh-3.2$
```

Observation:

Here we need to complete the function findPathToClosestDot and we are using the UCS to find that. As we have already implemented the UCSfunction earlier, we need only to call that function.

AutoGrader Result:

/Users/umangkumar/Desktop/NITD/Assignment/3rd/CSB 310/Labs/Lab4/PacmanGame/autograder.py:17: DeprecationWarning: the imp module is deprecated in favour of importlib; see the module's documentation for alternative uses
import imp
Starting on 9-24 at 23:54:29

Question q1
==========
***** PASS: test_cases/q1/graph_backtrack.test**
***** solution: ['1:A->C', '0:C->G']**
***** expanded_states: ['A', 'D', 'C']**
***** PASS: test_cases/q1/graph_bfs_vs_dfs.test**
***** solution: ['2:A->D', '0:D->G']**
***** expanded_states: ['A', 'D']**
***** PASS: test_cases/q1/graph_infinite.test**
***** solution: ['0:A->B', '1:B->C', '1:C->G']**
***** expanded_states: ['A', 'B', 'C']**
***** PASS: test_cases/q1/graph_manypaths.test**
***** solution: ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']**
***** expanded_states: ['A', 'B2', 'C', 'D', 'E2', 'F']**
***** PASS: test_cases/q1/pacman_1.test**

***** pacman layout: mediumMaze**
***** solution length: 130**
***** nodes expanded: 146**

Question q1: 3/3

Question q2
==========
***** PASS: test_cases/q2/graph_backtrack.test**
***** solution: ['1:A->C', '0:C->G']**
***** expanded_states: ['A', 'B', 'C', 'D']**
***** PASS: test_cases/q2/graph_bfs_vs_dfs.test**
***** solution: ['1:A->G']**
***** expanded_states: ['A', 'B']**
***** PASS: test_cases/q2/graph_infinite.test**
***** solution: ['0:A->B', '1:B->C', '1:C->G']**
***** expanded_states: ['A', 'B', 'C']**
***** PASS: test_cases/q2/graph_manypaths.test**
***** solution: ['1:A->C', '0:C->D', '1:D->F', '0:F->G']**
***** expanded_states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']**
***** PASS: test_cases/q2/pacman_1.test**
***** pacman layout: mediumMaze**
***** solution length: 68**
***** nodes expanded: 269**

Question q2: 3/3

Question q3
==========
***** PASS: test_cases/q3/graph_backtrack.test**
***** solution: ['1:A->C', '0:C->G']**
***** expanded_states: ['A', 'B', 'C', 'D']**
***** PASS: test_cases/q3/graph_bfs_vs_dfs.test**
***** solution: ['1:A->G']**
***** expanded_states: ['A', 'B']**
***** PASS: test_cases/q3/graph_infinite.test**
***** solution: ['0:A->B', '1:B->C', '1:C->G']**
***** expanded_states: ['A', 'B', 'C']**

```
*** PASS: test_cases/q3/graph_manypaths.test
***    solution:            ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***    expanded_states:     ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases/q3/ucs_0_graph.test
***    solution:            ['Right', 'Down', 'Down']
***    expanded_states:     ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q3/ucs_1_problemC.test
***    pacman layout:       mediumMaze
***    solution length: 68
***    nodes expanded:      269
*** PASS: test_cases/q3/ucs_2_problemE.test
***    pacman layout:       mediumMaze
***    solution length: 74
***    nodes expanded:      260
*** PASS: test_cases/q3/ucs_3_problemW.test
***    pacman layout:       mediumMaze
***    solution length: 152
***    nodes expanded:      173
*** PASS: test_cases/q3/ucs_4_testSearch.test
***    pacman layout:       testSearch
***    solution length: 7
***    nodes expanded:      14
*** PASS: test_cases/q3/ucs_5_goalAtDequeue.test
***    solution:            ['1:A->B', '0:B->C', '0:C->G']
***    expanded_states:     ['A', 'B', 'C']

### Question q3: 3/3 ###


Question q4
===========
*** PASS: test_cases/q4/astar_0.test
***    solution:            ['Right', 'Down', 'Down']
***    expanded_states:     ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q4/astar_1_graph_heuristic.test
***    solution:            ['0', '0', '2']
***    expanded_states:     ['S', 'A', 'D', 'C']
*** PASS: test_cases/q4/astar_2_manhattan.test
***    pacman layout:       mediumMaze
***    solution length: 68
```

***** nodes expanded: 221**
***** PASS: test_cases/q4/astar_3_goalAtDequeue.test**
***** solution: ['1:A->B', '0:B->C', '0:C->G']**
***** expanded_states: ['A', 'B', 'C']**
***** PASS: test_cases/q4/graph_backtrack.test**
***** solution: ['1:A->C', '0:C->G']**
***** expanded_states: ['A', 'B', 'C', 'D']**
***** PASS: test_cases/q4/graph_manypaths.test**
***** solution: ['1:A->C', '0:C->D', '1:D->F', '0:F->G']**
***** expanded_states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']**

Question q4: 3/3

Question q5
==========
***** PASS: test_cases/q5/corner_tiny_corner.test**
***** pacman layout: tinyCorner**
***** solution length: 28**

Question q5: 3/3

Question q6
==========
***** PASS: heuristic value less than true cost at start state**
***** PASS: heuristic value less than true cost at start state**
***** PASS: heuristic value less than true cost at start state**
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'West', 'West', 'West', 'South', 'South', 'South', 'West', 'West', 'North', 'East', 'East', 'North', 'North', 'North', 'North', 'East', 'East', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'West', 'West', 'South', 'South', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'East', 'East', 'North', 'North', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'North', 'North', 'East', 'East', 'North', 'North']
path length: 106
***** PASS: Heuristic resulted in expansion of 1136 nodes**

Question q6: 3/3

Question q7
==========
***** PASS: test_cases/q7/food_heuristic_1.test**
***** PASS: test_cases/q7/food_heuristic_10.test**
***** PASS: test_cases/q7/food_heuristic_11.test**
***** PASS: test_cases/q7/food_heuristic_12.test**
***** PASS: test_cases/q7/food_heuristic_13.test**
***** PASS: test_cases/q7/food_heuristic_14.test**
***** PASS: test_cases/q7/food_heuristic_15.test**
***** PASS: test_cases/q7/food_heuristic_16.test**
***** PASS: test_cases/q7/food_heuristic_17.test**
***** PASS: test_cases/q7/food_heuristic_2.test**
***** PASS: test_cases/q7/food_heuristic_3.test**
***** PASS: test_cases/q7/food_heuristic_4.test**
***** PASS: test_cases/q7/food_heuristic_5.test**
***** PASS: test_cases/q7/food_heuristic_6.test**
***** PASS: test_cases/q7/food_heuristic_7.test**
***** PASS: test_cases/q7/food_heuristic_8.test**
***** PASS: test_cases/q7/food_heuristic_9.test**
***** PASS: test_cases/q7/food_heuristic_grade_tricky.test**
***** expanded nodes: 4137**
***** thresholds: [15000, 12000, 9000, 7000]**

Question q7: 5/4

Question q8
==========
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_1.test**
***** pacman layout: Test 1**
***** solution length: 1**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_10.test**

***** pacman layout: Test 10**
***** solution length: 1**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_11.test**
***** pacman layout: Test 11**
***** solution length: 2**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_12.test**
***** pacman layout: Test 12**
***** solution length: 3**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_13.test**
***** pacman layout: Test 13**
***** solution length: 1**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_2.test**
***** pacman layout: Test 2**
***** solution length: 1**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_3.test**
***** pacman layout: Test 3**
***** solution length: 1**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_4.test**
***** pacman layout: Test 4**
***** solution length: 3**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_5.test**
***** pacman layout: Test 5**
***** solution length: 1**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_6.test**

***** pacman layout: Test 6**
***** solution length: 2**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_7.test**
***** pacman layout: Test 7**
***** solution length: 1**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_8.test**
***** pacman layout: Test 8**
***** solution length: 1**
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
***** PASS: test_cases/q8/closest_dot_9.test**
***** pacman layout: Test 9**
***** solution length: 1**

Question q8: 3/3

Finished at 23:54:47

Provisional grades
==========
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 5/4
Question q8: 3/3

Total: 26/25

Your grades are NOT yet registered. To register your grades, make sure to follow your instructor's guidelines to receive credit on your project.