ST1507: Data Structures and Algorithms

Assignment Two (CA2) AY22/23

**Group Report**

Lim Jun Jie (2100788)

Timothy Chia (2106911)

Group 7

DAAA/FT/2B/02

# 1. Description and User Guidelines

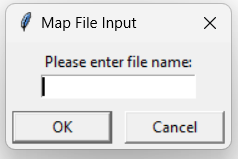
In this section we will have a brief description on how to start up and operate our application, as well as the user guidelines of this application.

## 1.a. Description

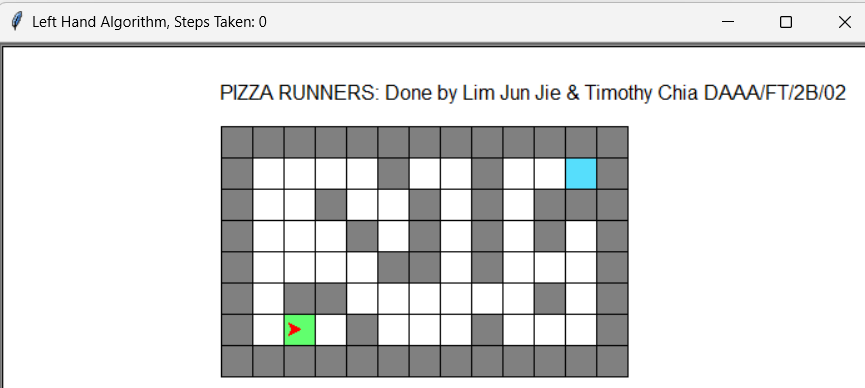
To start up this application, please open up the Anaconda Prompt and go into the directory where the project folder is located. Then, type “python main.py” into the terminal and the application will start up immediately.

## 1.b. User Guidelines

On start-up, the application will ask for a file name to open the input map from. The path set to open the map files from is the test\_files folder. Note that files with invalid input are not allowed and will be detected.

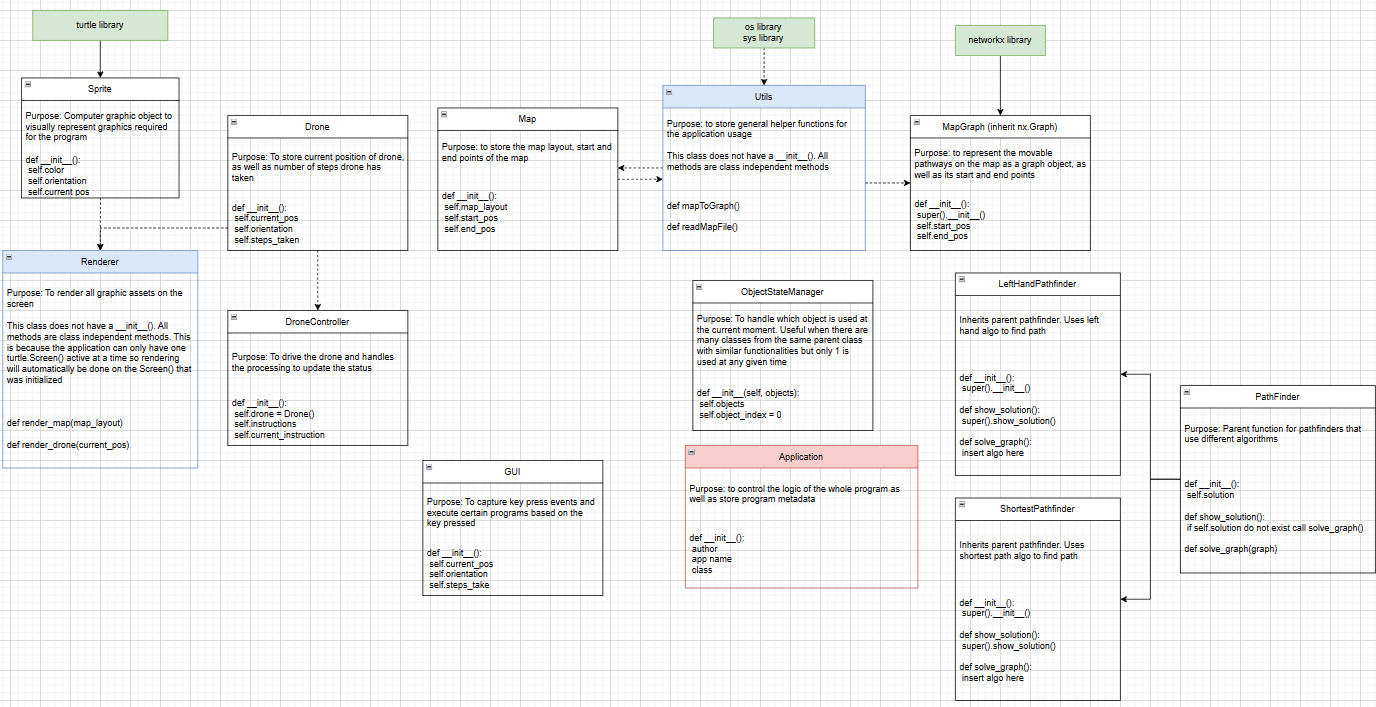


After the map has been loaded, the routes to the end point will be pre computed. Press ‘m’ key to move the drone and hold down ‘n’ key to move the drone continuously. Press tab to switch between algorithms. The current algorithm in use as well as the number of steps taken by the drone will be shown on the title bar, and the number of steps taken will be reset every time the algorithm being used is switched out.



# 2. Object-Oriented Programming Approach

In this section, we will talk more about how we made use of the Object-Oriented Programming (OOP) approach to design and create the application.



Above is the overall class diagram of our application.

The project is implemented using Object-Oriented Programming (OOP) to make use of the concepts:

1. Encapsulation
2. Polymorphism
3. Inheritance
4. Abstraction

## 2.a. Encapsulation

Encapsulation allows us to protect our class by hiding the internal implementation details and only exposing the variables we want to expose to the public. This allows us to control access to the data, making it possible to limit the ways in which the object’s data is modified.

In this application, encapsulation is applied in ALL class initialization variables by making them all private variables. These variables can be accessed through getter and setter functions, which will not be available for every private variable as not all private variables are exposed to the public.

However, there is an exception for the Pathfinder class, as I subclassed the Pathfinder class to form the LeftHandPathfinder and ShortestPathfinder classes, the variables in the parent class are protected instead of private, thus the child classes can access the class variables.

## 2.b. Polymorphism

Polymorphism was applied when working with the Pathfinder class. In order to keep the application source code reusable and scalable, we decided to create a parent class Pathfinder where pathfinders with different algorithms can inherit from. This allows us to use the same methods .solve() and .solution for all pathfinders, making all the pathfinder objects easier to use and reusable.

## 2.c. Inheritance

Inheritance was applied throughout when creating the application, with the obvious being the Pathfinder class that was made for the code to be more reusable.

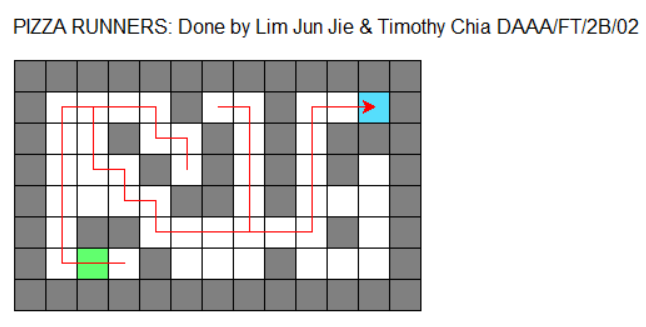
## 2.d. Abstraction

The entire application was made with abstraction in mind, which is the process of hiding the implementation details and showing only the essential features to the user. Abstraction is achieved through abstract classes, encapsulation and user interfaces, which are all implemented in the application through encapsulating all private variables and methods and only expose necessary ones, leveraging inheritance to make the code more reusable with Pathfinder class for example, and the use of a GUI class to capture key events for the program.

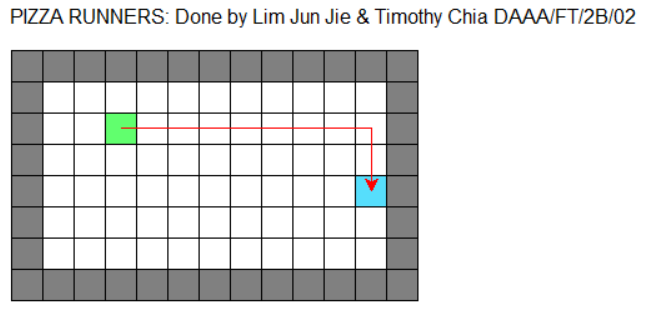
# 3. Data Structures and Algorithms Usage

In this section, we will talk more about the data structures and algorithms used or developed for this application, as well as the issues surrounding it.

## 3.a. Left Hand Rule Algorithm

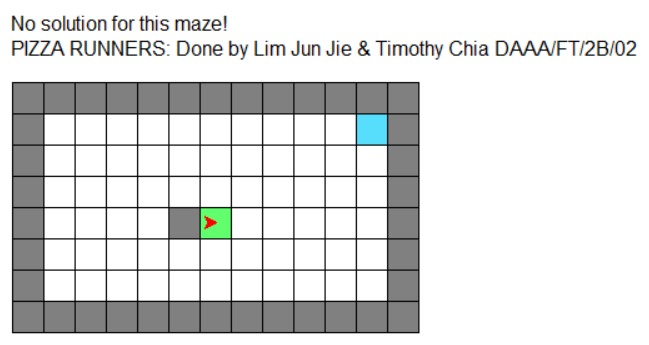


The left hand rule algorithm, also called the wall-follower algorithm, is an algorithm where the person’s left hand will always stick to the wall and not let go no matter what. The idea is that by always keeping the left hand in contact with the wall, the solver will eventually cover the whole maze and find the exit, as long as the maze is solvable.



However, the algorithm that we made has a special case, In the case where there is no walls to follow at the spawn, the solver will be made to walk forward till it reaches the wall, and when it has a wall in front of it it will turn to the right side, thus having the left hand touch the wall. This will guarantee a solution even when there is no wall to touch at spawn.

The other case is also where the solver will keep going left resulting in an infinite loop. For this, we will use a special case to check if it will be stuck in an infinite loop. If it will, we will let the solver walk forward to escape the loop, or turn right at a wall instead of left to escape the loop.



The left hand algorithm we made is not perfect, there is one test case this algorithm will not be able to solve, which is when the solver is in an empty space and there is only one block of wall. The solver will continuously loop around the wall if it is always made to go left first if available. Thus, if the algorithm has touched the spawn 4 times, it will end itself.

**Time Complexity:** O(n), this is the worst case where n is the number of available nodes to walk

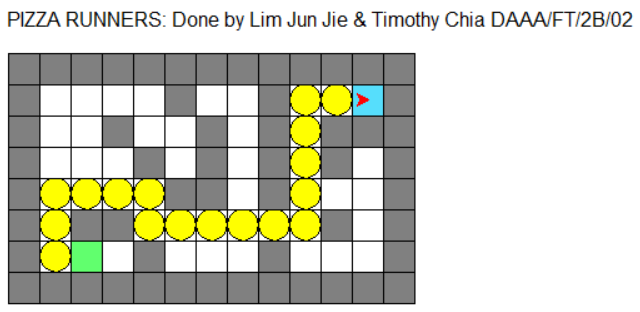
The algorithm will linearly search through the grid which means that the run time of the algorithm grows linearly with the size of the input.

**Space Complexity:** O(1)

The algorithm always uses a constant memory when calculating the algorithm, and thus the space complexity of this algorithm is constant

## 3.b. Shortest Path Algorithm

For the shortest path algorithm, we used Networkx Dijkstra’s and A Star (A\*) graph search algorithms to find the shortest path between start to end point.



Dijksta’s algorithm is a classic graph search algorithm that guarantees a shortest path between two nodes. It uses a priority queue underneath to keep track of the nodes to be processed. On the other hand A\* algorithm uses a heuristic function to guide the search in order to make it more efficient. This makes A\* faster than Djikstra’s most of the time when searching for the shortest path from point A to B. The heuristic function I used is Manhattan distance, due to its simplicity and ease of use.

For my Shortest Pathfinder, I decided to go with the Djikstra’s algorithm for mazes where number of walkable nodes are smaller than 50, and when it is larger the A\* will be used, as Djikstra’s algorithm is faster than the A\* algorithm for smaller graphs and unsolvable graphs while for larger graphs A\* will usually be faster.

**Time Complexity Dijkstra:** O(E + V log V), where E is number of edges and V is number of vertices

**Time Complexity A Star (A\*):** O(E + V log f(v)), where E is number of edges and V is number of vertices, f(v) is the value of the heuristic function

**Space Complexity:** O(1) for both

The algorithm always uses a constant memory when calculating the algorithm, and thus the space complexity of this algorithm is constant

## 3.c. Graph Structure (Networkx Graph)

We represented the layout of the paths of the map in a graph structure. This allows usage of graph algorithms to find the shortest paths in a maze. The graph we use is an undirected graph with a uniform weight of 1 for all edges.

## 3.d. Lists

We used many lists throughout the program to store necessary data as python lists allow for quick random access in O(1) time and are also easy to use and implement.

# 4. Challenges and Takeaways

In this section, we will talk about the challenges we faced while developing this application, as well as the take-aways and learning achievements.

## 4.a. Challenges

One big challenge that we faced is trying to optimize the left hand algorithm and handling the edge cases so that the algorithm is as optimized as possible. An example is that the left hand algo instead of doing nothing when there is no wall to touch, a way to optimize it is to walk forward to a wall to touch it instead of just not solving the maze.

Converting the shortest path solutions into coordinates for the turtle to move was another challenge, as we had to make sure that the turtle turned correctly and move to the correct coordinates at every step.

## 4.b. Takeaways

This project allowed us to deepen our understanding in graph algorithms and maze solving theory. We also got to apply Object Oriented Programming Concepts and this allowed us to practice the right way to apply OOP, learning from the mistakes we made in the first Assignment. Applying computational thinking and breaking huge tasks down into logical steps allowed us to have a clearer view on different solutions and allowed us to compare and weigh different solutions.

# 5. Roles and Contributions

# 

|  |  |
| --- | --- |
| Name | Tasks |
| Lim Jun Jie | 1. GUI class  2. Algorithm Switch  3. Left Hand Pathfinder  4. Renderer  5. Report  6. Write code comments |
| Timothy Chia | 1. Shortest Pathfinder  2. Turtle movement  3. Map generation  4. Utils class  5. Drone class  6. Write code comments |

# JJ and Tim

# Drone object to keep track of steps taken and execute instructions (the maze solution)

class Drone:

'''

Drone object to keep track of steps taken and execute instructions (the maze solution)

Parameters:

name (str): name of the drone

current\_pos (tuple): current position of the drone (x, y)

Attributes:

name (str): name of the drone

current\_pos (tuple): current position of the drone (x, y)

orientation (int): orientation of the drone (90 = North, 0 = East, 270 = South, 180 = West)

steps\_taken (int): number of steps taken by the drone

prev\_turn\_angle (int): angle of the last turn the drone made

'''

def \_\_init\_\_(self, name, current\_pos):

self.\_\_name = name

self.\_\_start\_pos = current\_pos

self.\_\_current\_pos = current\_pos

self.\_\_orientation = 0 # Legend for orientation: 90 = North, 0 = East, 270 = South, 180 = West (turtle orientations)

self.\_\_prev\_turn\_angle = 0 # track the last turn angle the drone made.

self.\_\_steps\_taken = 0 # track the number of steps taken by the drone: only +1 when it moves. +0 when it turns

@property

def name(self):

return self.\_\_name

@property

def current\_pos(self):

return self.\_\_current\_pos

@property

def orientation(self):

return self.\_\_orientation

@property

def steps\_taken(self):

return self.\_\_steps\_taken

@property

def prev\_turn\_angle(self):

return self.\_\_prev\_turn\_angle

@name.setter

def name(self, value):

self.\_\_name = value

@current\_pos.setter

def current\_pos(self, value):

self.\_\_current\_pos = value

@orientation.setter

def orientation(self, value):

self.\_\_orientation = value

@steps\_taken.setter

def steps\_taken(self, value):

self.\_\_steps\_taken = value

@prev\_turn\_angle.setter

def prev\_turn\_angle(self, value):

self.\_\_prev\_turn\_angle = value

def reset(self):

'''Reset the drone to its starting position and orientation'''

self.\_\_current\_pos = self.\_\_start\_pos

self.\_\_orientation = 0

self.\_\_prev\_turn\_angle = 0

self.\_\_steps\_taken = 0

# execute instructions to drive the drone. Controller is unique to each drone

# JJ and Tim

class DroneController:

'''

Controller to execute instructions (the maze solution) for the drone.

Parameters:

drone (Drone): drone object to be controlled

instructions (list): list of instructions to be executed by the drone

Attributes:

drone (Drone): drone object to be controlled

instructions (list): list of instructions to be executed by the drone

current\_instruction (int): index of the current instruction to be executed

'''

def \_\_init\_\_(self, drone, instructions=None):

self.\_\_drone = drone

self.\_\_instructions = instructions

self.\_\_current\_instruction = 0

@property

def instructions(self):

return self.\_\_instructions

@property

def existing\_instructions(self):

return self.\_\_instructions[self.\_\_current\_instruction:]

@instructions.setter

def instructions(self, value):

self.\_\_instructions = value

self.\_\_current\_instruction = 0

# move drone and update drone status

def execute\_instruction(self):

'''

Move drone and update drone status.

Returns:

bool: True if there are no more instructions to be executed, False otherwise

'''

# check if there are still instructions to be executed

if self.\_\_current\_instruction < len(self.\_\_instructions):

# get the next instruction

instruction = self.\_\_instructions[self.\_\_current\_instruction]

self.\_\_current\_instruction += 1

# check if the instruction is a turn or a move

if type(instruction) == list:

self.\_\_drone.prev\_turn\_angle = instruction[0]

self.\_\_drone.orientation = instruction[1]

elif type(instruction) == tuple: # tuple means its a move

self.\_\_drone.current\_pos = instruction

self.\_\_drone.steps\_taken = self.\_\_drone.steps\_taken + 1 # update the number of steps taken by the drone

else:

return True

import turtle

from classes.utils import Utils

from classes.mazegenerator import MazeGenerator

from classes.map import Map

# JJ and Tim

class GUI:

'''

Handles all GUI related events

Parameters:

screen (turtle.Screen): the screen that the GUI is connected to

Attributes:

\_\_moveDrone (bool): whether or not the drone should move

\_\_changeAlgo (bool): whether or not the algorithm should change

screen (turtle.Screen): the screen that the GUI is connected to

'''

def \_\_init\_\_(self, screen):

self.\_\_moveDrone = False

self.\_\_changeAlgo = False

self.\_\_activate\_random\_obstacles = False

self.\_\_generate\_maze = False

self.\_\_activate\_tsp = False

self.\_\_screen = screen # gui NEEDS to know what screen it is connected to so that it can call the screen's ontimer method to capture events

def command\_move\_drone(self):

self.\_\_moveDrone = True

def command\_change\_algorithm(self):

self.\_\_changeAlgo = True

def command\_activate\_random\_maze(self):

self.\_\_generate\_maze = True

def command\_activate\_random\_obstacles(self):

if self.\_\_activate\_random\_obstacles == False:

self.\_\_activate\_random\_obstacles = True

else:

self.\_\_activate\_random\_obstacles = False

def command\_activate\_tsp(self):

if self.\_\_activate\_tsp == False:

self.\_\_activate\_tsp = True

# extra feature

def tsp\_event\_listener(self, drone\_controller, renderer, drone, tsp\_pathfinder):

if self.\_\_activate\_tsp == True:

# read the map graph and create a new map just for TSP

# this is because the map graph will have multiple end points

# create new Map just for TSP

map\_layout, start, ends = Utils.readMapFile("map\_files/tsp.txt", multiple\_end\_points=True)

tsp\_map = Map(map\_layout, start, ends)

# map to graph

map\_graph = Utils.mapToGraph(tsp\_map)

# change the start pos of map graph to the current position of the drone

solution = tsp\_pathfinder.solve(map\_graph, update\_solution=True, drone\_orientation=drone.orientation)

drone\_controller.instructions = solution

# reset drone

drone.reset()

# rendering stuff

renderer.render\_map(tsp\_map)

renderer.clear\_guidelight() # clear guidelight if any

renderer.render\_drone(drone, spawn=True) # spawn drone

# rendering stuff

renderer.clear\_guidelight()

def move\_drone\_event\_listener(self, drone\_controller, renderer, drone, pathfinders, terrain\_changer, map\_graph):

'''

Event listener for the move drone command

Parameters:

drone\_controller (DroneController): controller to execute instructions for the drone

render\_drone (function): function to render the drone

drone (Drone): drone object to be controlled

pathfinders (Pathfinder): Pathfinder object that contains all the algorithms

Returns:

None

'''

if self.\_\_moveDrone == True and drone.current\_pos != map\_graph.end\_pos:

if self.\_\_changeAlgo == False:

# tell drone controller to execute the next instruction

drone\_controller.execute\_instruction()

renderer.render\_drone(drone)

# extra feature to add random obstacles

if self.\_\_activate\_random\_obstacles == True and drone.current\_pos != map\_graph.end\_pos:

# change road to wall and remove the node from graph

node\_info = terrain\_changer.road\_to\_wall(drone.current\_pos)

map\_graph.remove\_node(node\_info[0])

solution = pathfinders.current.solution

# check if node removed exists in the solution

if node\_info[0] in solution:

# change the start pos of map graph to the current position of the drone

map\_graph.start\_pos = drone.current\_pos

solution = pathfinders.current.solve(map\_graph, update\_solution=True, drone\_orientation=drone.orientation)

drone\_controller.instructions = solution

# rendering stuff

renderer.clear\_guidelight() # clear guidelight if any

# check if there is guidelight to be rendered

if pathfinders.current.id == 1 and solution != []:

renderer.render\_guidelight(solution)

# if drone not at end

if drone.current\_pos != map\_graph.end\_pos:

renderer.render\_no\_solution(solution)

# render the updated block

renderer.render\_map\_block(node\_info[0][0], node\_info[0][1], node\_info[1])

# update the title bar

self.\_\_screen.title(f"{pathfinders.current.algorithm\_name}, Steps Taken: {drone.steps\_taken}")

self.\_\_moveDrone = False # set to false

self.\_\_screen.ontimer(lambda: self.move\_drone\_event\_listener(drone\_controller, renderer, drone, pathfinders, terrain\_changer, map\_graph), 10)

def change\_algorithm\_event\_listener(self, drone\_controller, renderer, drone, pathfinders, map\_graph):

'''

Event listener for the change algorithm command

Parameters:

drone\_controller (DroneController): controller to execute instructions for the drone

render\_drone (function): function to render the drone

drone (Drone): drone object to be controlled

pathfinders (Pathfinder): Pathfinder object that contains all the algorithms

map\_graph (Graph): Graph object that contains the map

Returns:

None

'''

if self.\_\_changeAlgo == True:

# change current algorithm

pathfinders.next()

# reset drone

drone.reset()

map\_graph.start\_pos = drone.current\_pos

solution = pathfinders.current.solve(map\_graph, update\_solution=True)

drone\_controller.instructions = solution # reset drone controller with new instructions

renderer.render\_no\_solution(solution)

# display the guidelight for shortest path algorithm

if pathfinders.current.id == 1:

# render guidelight for solution if shortest path is being used

renderer.clear\_dronetrail(drone.name)

renderer.render\_guidelight(solution)

# respawn drone sprite at start

renderer.render\_drone(drone, spawn=True)

else:

renderer.clear\_guidelight()

# respawn drone sprite at start

renderer.render\_drone(drone, spawn=True)

if solution != []:

renderer.render\_dronetrail(drone.name)

# reset title bar and boolean

self.\_\_screen.title(f"{pathfinders.current.algorithm\_name}, Steps Taken: {drone.steps\_taken}")

self.\_\_changeAlgo = False

self.\_\_screen.ontimer(lambda: self.change\_algorithm\_event\_listener(drone\_controller, renderer, drone, pathfinders, map\_graph), 10)

def random\_maze\_event\_listener(self):

if self.\_\_generate\_maze == True:

size = self.get\_input("Generate Maze", "Enter the size of the maze (e.g. 10): ")

size = int(size)

maze = MazeGenerator(size, size)

maze.\_\_generate\_maze(0, 0)

maze.write\_to\_file()

self.\_\_generate\_maze = False

self.\_\_screen.ontimer(lambda: self.random\_maze\_event\_listener(), 10)

def get\_input(self, title, prompt):

result = turtle.textinput(title, prompt)

return result

import networkx as nx

from classes.pathfinder import Pathfinder

# inherit pathfinder class

class LeftHandPathfinder(Pathfinder):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.\_algorithm\_name = "Left Hand Algorithm"

self.\_id = 0

def solve(self, map\_graph, update\_solution=False, drone\_orientation=None):

if self.\_solution != None and update\_solution == False:

return self.\_solution

elif nx.has\_path(map\_graph, map\_graph.start\_pos, map\_graph.end\_pos) == False:

return []

else:

# keep track of how many times we visited start node. if more than 4 times means we stuck in infinite loop

start\_pos = map\_graph.start\_pos

visited\_start\_counter = 0

x, y = map\_graph.start\_pos

# array to store steps

route\_instructions = []

# tuples are in order of left, forward, right, back according to current orientation

facingNorth = [(-1, 0), (0, 1), (1, 0), (0, -1)]

facingEast = [(0, 1), (1, 0), (0, -1), (-1, 0)]

facingSouth = [(1, 0), (0, -1), (-1, 0), (0, 1)]

facingWest = [(0, -1), (-1, 0), (0, 1), (1, 0)]

# keep track of orientation when moving through maze

# North, East, South, West. according to turtle orientations

orientation = [90, 0, 270, 180]

compass = [facingNorth, facingEast, facingSouth, facingWest]

check\_diagonals = [(-1,-1), (-1, 1), (1, 1), (1, -1)]

# keep track of how to turn the compass and the direction

# ordered by left, forward, right, back

turn = [-1, 0, 1, 2]

turn\_angle = [-90, 0, 90, 180]

# if not speified, set starting orientation to east (index of 1), this is default orientation the pen will face in turtle when it first starts

# keep track of index in compass array, so that we can get index in O(1) time instead of using .index() which is O(n)

# when changing index, always mod by 4 to keep it within range

if drone\_orientation == None:

index = 1

else:

index = orientation.index(drone\_orientation)

# keep finding the next node to go to until we reach the end

while (x, y) != map\_graph.end\_pos:

# get current orientation

current\_orientation = compass[index]

# step 1: check if left can, if can, check if infinite loop exists

# index 0 means left side, 2nd indexing is to get the tuple x and y values

left\_x, left\_y = x + current\_orientation[0][0], y + current\_orientation[0][1]

if map\_graph.has\_node((left\_x, left\_y)):

# to check if infinite loop: check the diagonal and left and bottom

# if it is a infinite loop, check if there is a wall in front

# if wall in front, turn right so the left hand touches the wall

# else walk forward to escape the loop

bottom\_x, bottom\_y = x + current\_orientation[3][0], y + current\_orientation[3][1]

diagonal\_x, diagonal\_y = x + check\_diagonals[index][0], y + check\_diagonals[index][1]

if map\_graph.has\_node((bottom\_x, bottom\_y)) and map\_graph.has\_node((diagonal\_x, diagonal\_y)):

# if its not a wall, go forward, else go to the right

front\_x, front\_y = x + current\_orientation[1][0], y + current\_orientation[1][1]

if map\_graph.has\_node((front\_x, front\_y)):

x, y = front\_x, front\_y

route\_instructions.append((x, y))

else:

# update the new orientation, turn right

index = (index + turn[2]) % 4

# store the new orientation of the drone, as well as its turn angle

route\_instructions.append([turn\_angle[2], orientation[index]])

# no infinite loop, drone can go to the left safely

else:

# update the new orientation, turn left means index 0

index = (index + turn[0]) % 4

# store the new orientation of the drone, as well as its turn angle

route\_instructions.append([turn\_angle[0], orientation[index]])

# update current position and add new position to route instructions

x, y = left\_x, left\_y

route\_instructions.append((x, y))

# we need to check every direction, starting from forward > right > back

else:

for direction in range(1, 4):

# check if the next node in the direction exists

x\_next, y\_next = x + current\_orientation[direction][0], y + current\_orientation[direction][1]

if map\_graph.has\_node((x\_next, y\_next)):

if direction == 3: # if we turned back, we need to turn twice, so turn right 2 times

# update the new orientation according to the direction we turned

index = (index + turn[2]) % 4

route\_instructions.append([turn\_angle[2], orientation[index]])

index = (index + turn[2]) % 4

route\_instructions.append([turn\_angle[2], orientation[index]])

else:

# update the new orientation according to the direction we turned

index = (index + turn[direction]) % 4

if direction != 1: # if not forward, add turning to route instructions

route\_instructions.append([turn\_angle[direction], orientation[index]])

# update position and add to route instructions

x, y = x\_next, y\_next

route\_instructions.append((x, y))

# break out of loop as we found the next node

break

# check if we visited the start node

if (x, y) == start\_pos:

visited\_start\_counter += 1

if visited\_start\_counter == 4:

# impossible to escape, exit program

return []

# set solution to route instructions

self.\_solution = route\_instructions

print(route\_instructions)

print("Calculated Left Hand Algo")

return route\_instructions

# map object, stores necessary metadata for the map

class Map:

'''

Stores map layout, start position, end position, and map dimensions

Parameters:

map\_layout (list): 2D array of 0s and 1s, 0s represent walls, 1s represent open spaces

start\_pos (tuple): tuple of x and y coordinates of the start position

end\_pos (tuple): tuple of x and y coordinates of the end position

Attributes:

layout (list): 2D array of 0s and 1s, 0s represent walls, 1s represent open spaces

start\_pos (tuple): tuple of x and y coordinates of the start position

end\_pos (tuple): tuple of x and y coordinates of the end position

x\_length (int): horizontal length of the map in terms of number of cells

y\_length (int): vertical length of the map in terms of number of cells

'''

def \_\_init\_\_(self, map\_layout, start\_pos, end\_pos):

self.\_\_layout = map\_layout

self.\_\_start\_pos = start\_pos

self.\_\_end\_pos = end\_pos

# horizontal\_length (x\_length) and vertical\_length (y\_length) of the map in terms of number of cells

self.\_\_x\_length = len(map\_layout[0])

self.\_\_y\_length = len(map\_layout)

# getter functions

@property

def layout(self):

return self.\_\_layout

@property

def start\_pos(self):

return self.\_\_start\_pos

@property

def end\_pos(self):

return self.\_\_end\_pos

@property

def x\_length(self):

return self.\_\_x\_length

@property

def y\_length(self):

return self.\_\_y\_length

@layout.setter

def layout(self, value):

self.\_\_layout = value

# map can create its own obstacles by changing its own layout

import networkx as nx

# subclass nx graph

class MapGraph(nx.Graph):

'''

A graph representing the map of the game.

Parameters:

map\_layout (list): 2D array of 0s and 1s, 0s represent walls, 1s represent open spaces

start\_pos (tuple): tuple of x and y coordinates of the start position

end\_pos (tuple): tuple of x and y coordinates of the end position

Attributes:

layout (list): 2D array of 0s and 1s, 0s represent walls, 1s represent open spaces

start\_pos (tuple): tuple of x and y coordinates of the start position

end\_pos (tuple): tuple of x and y coordinates of the end position

'''

def \_\_init\_\_(self, start\_pos, end\_pos):

super().\_\_init\_\_()

self.\_\_start\_pos = start\_pos

self.\_\_end\_pos = end\_pos

@property

def start\_pos(self):

return self.\_\_start\_pos

@property

def end\_pos(self):

return self.\_\_end\_pos

@start\_pos.setter

def start\_pos(self, value):

self.\_\_start\_pos = value

@end\_pos.setter

def end\_pos(self, value):

self.\_\_end\_pos = value

import os

import random

class MazeGenerator:

'''

Class to generate a maze with DFS algorithm.

Parameters:

width (int): width of the maze

height (int): height of the maze

Attributes:

width (int): width of the maze

height (int): height of the maze

maze (list): 2D array of 0s and 1s, 0s represent walls, 1s represent open spaces

visited (list): 2D array of booleans, True if the cell has been visited, False otherwise

directions (list): list of directions to move in

dx (list): list of x coordinates of the directions

dy (list): list of y coordinates of the directions

start (tuple): tuple of x and y coordinates of the start position

end (tuple): tuple of x and y coordinates of the end position

'''

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

self.maze = [[1 for \_ in range(width)] for \_ in range(height)]

self.visited = [[False for \_ in range(width)] for \_ in range(height)]

self.directions = ["N", "S", "E", "W"]

self.dx = [0, 0, 1, -1]

self.dy = [-1, 1, 0, 0]

# random start and end

self.start = None

self.end = None

def generate\_maze(self, x, y):

'''

Maze is stored in self.maze attribute.

Parameters:

x (int): x coordinate of the current cell

y (int): y coordinate of the current cell

Returns:

None

'''

self.visited[y][x] = True

random.shuffle(self.directions)

for direction in self.directions:

if direction == "N":

if y > 1 and not self.visited[y-2][x]:

self.maze[y-1][x] = 0

self.maze[y-2][x] = 0

self.generate\_maze(x, y-2)

elif direction == "S":

if y < self.height-2 and not self.visited[y+2][x]:

self.maze[y+1][x] = 0

self.maze[y+2][x] = 0

self.generate\_maze(x, y+2)

elif direction == "E":

if x < self.width-2 and not self.visited[y][x+2]:

self.maze[y][x+1] = 0

self.maze[y][x+2] = 0

self.generate\_maze(x+2, y)

elif direction == "W":

if x > 1 and not self.visited[y][x-2]:

self.maze[y][x-1] = 0

self.maze[y][x-2] = 0

self.generate\_maze(x-2, y)

if not self.start:

self.start = (x, y)

else:

self.end = (x, y)

def write\_to\_file(self, filename='map\_files/random\_maze.txt'):

# write to file

with open(filename, "w") as f:

for y, row in enumerate(self.maze):

for x, cell in enumerate(row):

if (x, y) == self.start:

f.write("s")

elif (x, y) == self.end:

f.write("e")

elif cell == 1:

f.write("X")

else:

f.write(".")

f.write("\n")

# this class is needed in order to cycle through multiple objects

class ObjectStateManager:

'''

This class is used to cycle through multiple objects

Parameters:

objects (list): list of objects

Attributes:

objects (list): list of objects

object\_index (int): index of the current object

'''

def \_\_init\_\_(self, objects):

self.\_\_objects = objects

self.\_\_object\_index = 0

@property

def current(self):

return self.\_\_objects[self.\_\_object\_index]

def next(self):

self.\_\_object\_index = (self.\_\_object\_index + 1) % len(self.\_\_objects) # modulo will help us to cycle through the different objects

# Parent class pathfinder

class Pathfinder:

'''

This is the parent class for all pathfinding algorithms.

Attributes:

solution (list): list of tuples representing the path from start to end

algorithm\_name (str): name of the algorithm

'''

def \_\_init\_\_(self):

self.\_solution = None

self.\_algorithm\_name = None

self.\_id = None

@property

def algorithm\_name(self):

return self.\_algorithm\_name

@property

def solution(self):

return self.\_solution

@property

def id(self):

return self.\_id

def solve(self, graph, update\_solution=False):

'''

This method must be implemented in a subclass.

Parameters:

graph (Graph): graph object

Returns:

None

'''

raise NotImplementedError("This method must be implemented in a subclass")

import turtle

# renderer class to render graphics object on the screen

class Renderer:

def \_\_init\_\_(self, pixel\_size):

self.\_\_graphics\_assets = dict()

self.\_\_pixel\_size = pixel\_size # pixel size of each unit space on the map

# calculate how much to offset the map so that it will be centered on the screen

self.\_\_offset\_x = None

self.\_\_offset\_y = None

# keep the max height of the map so we can render words above

self.\_\_map\_max\_height = None

@property

def pixel\_size(self):

return self.\_\_pixel\_size

# store the graphic representation of the sprite in the dictionary

def add\_sprite(self, sprite):

self.\_\_graphics\_assets[sprite.name] = sprite

# re render this on every new algo

def render\_no\_solution(self, solution):

writer = self.\_\_graphics\_assets['writer']

writer.clear()

if solution == []:

writer.goto(-self.\_\_offset\_x - (self.\_\_pixel\_size / 2), self.\_\_map\_max\_height + 25)

writer.write("No solution for this maze!", font=("Arial", 12, "normal"))

def render\_title(self, title):

# create a new turtle object to draw the title

title\_turtle = turtle.Turtle()

title\_turtle.hideturtle()

title\_turtle.penup()

title\_turtle.color("black")

title\_turtle.goto(-self.\_\_offset\_x - (self.\_\_pixel\_size / 2), self.\_\_map\_max\_height + 5)

title\_turtle.write(title, font=("Arial", 12, "normal"))

def render\_map(self, map):

'''

Renders the given map input on the screen. The map

must be an array of strings, with each string representing

a row of the map. Each character in the string represents

a block in the map. The following characters are used:

- X: Wall

- s: Start point

- e: End point

- .: Normal road

'''

# calculate offset

self.\_\_calculate\_offset(map.x\_length, map.y\_length)

# calculate max height

self.\_\_calculate\_map\_max\_height(map.y\_length)

# get all the sprites

wall = self.\_\_graphics\_assets["wall"]

road = self.\_\_graphics\_assets["road"]

start = self.\_\_graphics\_assets["startpoint"]

end = self.\_\_graphics\_assets["endpoint"]

# get map layout

map\_layout = map.layout

### draw the map by placing tiles according to map layout

for y\_pos in range(len(map\_layout)): # y axis of map, starting from top (according to map layout in text file)

for x\_pos in range(len(map\_layout[y\_pos])): # x axis of map starting from left

# reverse y\_pos to start from bottom

y\_reversed = len(map\_layout) - y\_pos - 1

# calculate position of grid

grid\_x\_pos = self.\_\_pixel\_size \* x\_pos - self.\_\_offset\_x

grid\_y\_pos = self.\_\_pixel\_size \* y\_reversed - self.\_\_offset\_y

# check if it is a wall

if map\_layout[y\_pos][x\_pos] == "X":

block = wall

# check if it is a road

elif map\_layout[y\_pos][x\_pos] == ".":

block = road

# check if it is start point

elif map\_layout[y\_pos][x\_pos] == "s":

block = start

# it is a end point

else:

block = end

# configure the tile drawing``

block.goto((grid\_x\_pos, grid\_y\_pos))

# place down tile on the screen

block.stamp()

def reset\_drone(self, drone):

drone\_sprite = self.\_\_graphics\_assets[drone.name]

drone\_sprite.reset()

def render\_drone(self, drone, spawn=False):

'''

Updates the location and orientation given DroneSprite object on the screen.

'''

### Update its location and orientation

drone\_sprite = self.\_\_graphics\_assets[drone.name]

# if spawn is true, renders drone sprite immediately at the current position of drone

if spawn:

drone\_sprite.hideturtle()

# move the drone to the starting position

drone\_sprite.goto(drone.current\_pos[0] \* self.\_\_pixel\_size - self.\_\_offset\_x, drone.current\_pos[1] \* self.\_\_pixel\_size - self.\_\_offset\_y)

# change the orientation of the drone

drone\_sprite.setheading(0)

# update the current position of the drone

drone\_sprite.current\_pos = drone.current\_pos

# update the orientation of the drone

drone\_sprite.orientation = drone.orientation

# show the drone

drone\_sprite.showturtle()

# check if orientation changed

elif drone.orientation != drone\_sprite.orientation:

drone\_sprite.right(drone.prev\_turn\_angle)

# update the new orientation of the drone

drone\_sprite.orientation = drone.orientation

# if orientation did not change, means the position changed

elif drone.current\_pos != drone\_sprite.current\_pos:

drone\_sprite.goto(drone.current\_pos[0] \* self.\_\_pixel\_size - self.\_\_offset\_x, drone.current\_pos[1] \* self.\_\_pixel\_size - self.\_\_offset\_y)

# update the current position of the drone

drone\_sprite.current\_pos = drone.current\_pos

def render\_guidelight(self, solution):

# load the guide light sprite

guide\_light = self.\_\_graphics\_assets["guidelight"]

for i in range(len(solution) - 1): # minus 1 because we dont want the guidelight to cover up the end point

if type(solution[i]) == tuple:

guide\_light.goto(solution[i][0] \* self.\_\_pixel\_size - self.\_\_offset\_x, solution[i][1] \* self.\_\_pixel\_size - self.\_\_offset\_y)

guide\_light.stamp()

def clear\_guidelight(self):

guide\_light = self.\_\_graphics\_assets["guidelight"]

guide\_light.clear()

def render\_dronetrail(self, drone\_name):

drone\_sprite = self.\_\_graphics\_assets[drone\_name]

drone\_sprite.pendown()

def clear\_dronetrail(self, drone\_name):

drone\_sprite = self.\_\_graphics\_assets[drone\_name]

drone\_sprite.clear()

drone\_sprite.penup()

def render\_map\_block(self, x, y, block\_type):

# render a block on the map

if block\_type == "wall":

block = self.\_\_graphics\_assets["wall"]

block.goto(x \* self.\_\_pixel\_size - self.\_\_offset\_x, y \* self.\_\_pixel\_size - self.\_\_offset\_y)

block.stamp()

def \_\_calculate\_offset(self, map\_x\_length, map\_y\_length):

'''

Calculates the offset of the map so that it will be centered on the screen.

'''

# calculate the offset

self.\_\_offset\_x = (map\_x\_length \* self.\_\_pixel\_size) / 2

self.\_\_offset\_y = (map\_y\_length \* self.\_\_pixel\_size) / 2

def \_\_calculate\_map\_max\_height(self, map\_y\_length):

self.\_\_map\_max\_height = map\_y\_length \* self.\_\_pixel\_size - self.\_\_offset\_y

import networkx as nx

from classes.pathfinder import Pathfinder

# inherit pathfinder class

class ShortestPathfinder(Pathfinder):

'''

This class implements the shortest path algorithm.

Attributes:

algorithm\_name (str): name of the algorithm

'''

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.\_algorithm\_name = "Shortest Path Algorithm"

self.\_id = 1

# use manhattan distance as heuristic

self.\_\_heuristic = lambda u, v: abs(u[0] - v[0]) + abs(u[1] - v[1])

def solve(self, map\_graph, update\_solution=False, drone\_orientation=None):

'''

This method finds the shortest path from the start node to the end node.

Parameters:

map\_graph (Graph): graph object

Returns:

None

'''

if self.\_solution != None and update\_solution == False:

return self.\_solution

elif nx.has\_path(map\_graph, map\_graph.start\_pos, map\_graph.end\_pos):

if map\_graph.number\_of\_nodes() < 50:

print("Using Dijkstra's algorithm")

path = nx.shortest\_path(map\_graph, map\_graph.start\_pos, map\_graph.end\_pos) # djiksra's algorithm

else:

print("Using A\* algorithm")

path = nx.astar\_path(map\_graph, map\_graph.start\_pos, map\_graph.end\_pos, heuristic=self.\_\_heuristic)

# convert path to route instructions

route\_instructions = []

facingNorth = [(-1, 0), (0, 1), (1, 0), (0, -1)]

facingEast = [(0, 1), (1, 0), (0, -1), (-1, 0)]

facingSouth = [(1, 0), (0, -1), (-1, 0), (0, 1)]

facingWest = [(0, -1), (-1, 0), (0, 1), (1, 0)]

turn\_angle = [-90, 0, 90, 180]

turn = [-1, 0, 1, 2]

# North, East, South, West. according to turtle orientations

orientation = [90, 0, 270, 180]

compass = [facingNorth, facingEast, facingSouth, facingWest]

if drone\_orientation == None:

index = 1 # default orientation is east

else:

index = orientation.index(drone\_orientation)

for i in range(len(path) - 1): # -1 as we don't need to check the last node

current\_direction = compass[index] # default orientation is east for turtle

# get the x and y difference between 2 nodes

x\_diff = path[i + 1][0] - path[i][0]

y\_diff = path[i + 1][1] - path[i][1]

# find new orientation

next\_direction\_index = current\_direction.index((x\_diff, y\_diff)) # Worst case O(4)

if next\_direction\_index != 1: # if not forward, get turn angle

if next\_direction\_index == 3: # if back turn 90 degree 2 times

index = (index + turn[2]) % 4

route\_instructions.append([turn\_angle[2], orientation[index]])

index = (index + turn[2]) % 4

route\_instructions.append([turn\_angle[2], orientation[index]])

else:

index = (index + turn[next\_direction\_index]) % 4

route\_instructions.append([turn\_angle[next\_direction\_index], orientation[index]])

route\_instructions.append(path[i + 1])

# set solution to route instructions

self.\_solution = route\_instructions

print(route\_instructions)

print("Calculated Shortest Path Algo")

return route\_instructions

else:

return []

import turtle

class Sprite(turtle.Turtle):

'''

Sprite class that inherits from Turtle class.

Parameters:

name (str): name of the sprite

color (str): color of the sprite

pencolor (str): color of the pen

shape (str): shape of the sprite

shapesize (int): size of the sprite

speed (int): speed of the sprite

current\_pos (tuple): current position of the sprite

orientation (str): orientation of the sprite

Attributes:

name (str): name of the sprite

current\_pos (tuple): current position of the sprite

orientation (str): orientation of the sprite

'''

def \_\_init\_\_(self, name, color, pencolor=None, shape=None, shapesize=None, speed=None, current\_pos=None, orientation=None):

super().\_\_init\_\_()

self.hideturtle()

self.penup()

# cannot set None when setting colors

if color != None:

self.color(color)

if pencolor != None:

self.pencolor(pencolor)

self.shape(shape)

self.shapesize(shapesize) # 1 unit = 20 pixels for shapesize()

self.speed(speed)

self.\_\_name = name

self.\_\_current\_pos = current\_pos

self.\_\_orientation = orientation

@property

def name(self):

return self.\_\_name

@property

def orientation(self):

return self.\_\_orientation

@property

def current\_pos(self):

return self.\_\_current\_pos

@name.setter

def name(self, value):

self.\_\_name = value

@orientation.setter

def orientation(self, value):

self.\_\_orientation = value

@current\_pos.setter

def current\_pos(self, value):

self.\_\_current\_pos = value

import random

# class to change the map terrain

class TerrainChanger:

def \_\_init\_\_(self, map, map\_graph): # takes in map and map\_graph object

self.\_\_map = map

self.\_\_map\_graph = map\_graph

def road\_to\_wall(self, drone\_current\_pos):

# get random node

if len(self.\_\_map\_graph.nodes()) > 3:

# get random node but not the start or end node, or the current position of the drone

node = random.choice([node for node in self.\_\_map\_graph.nodes() if node != self.\_\_map.start\_pos and node != self.\_\_map.end\_pos and node != drone\_current\_pos])

x, y = node

# remove node from graph as it is now a wall

# self.\_\_map\_graph.remove\_node(node)

# change map layout

# reverse y\_pos to start from bottom

y\_reversed = self.\_\_map.y\_length - y - 1

self.\_\_map.layout[y\_reversed][x] = "X"

# return the node that was changed

return [node, "wall"]

import networkx as nx

from classes.pathfinder import Pathfinder

# inherit pathfinder class

class TravellingSalesmanPathfinder(Pathfinder):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.\_algorithm\_name = "Travelling Salesman Problem Pathfinder"

self.\_id = 2

def solve(self, map\_graph):

if self.\_solution != None:

return self.\_solution

else:

tsp = nx.approximation.traveling\_salesman\_problem

map\_graph.end\_pos.insert(0, map\_graph.start\_pos)

path = tsp(map\_graph, nodes=map\_graph.end\_pos)

return path

import os

import sys

from classes.mapgraph import MapGraph

# class to store general functions to support the application

class Utils:

'''

This class contains general functions to support the application.

'''

# takes in the processed map text and converts it to a graph

def map\_to\_graph(map):

'''

This function takes in the processed map text and converts it to a graph.

Parameters:

map (Map): map object

Returns:

graph (MapGraph): graph object

'''

# Create an empty graph

graph = MapGraph(map.start\_pos, map.end\_pos)

# Add the nodes to the graph

# nodes are the coordinates of each road tile in the map

for y in range(len(map.layout)): # go from bottom up

for x in range(len(map.layout[0])):

if map.layout[y][x] != 'X': #1 is used to represent walls

# reverse y axis order before adding node

y\_reverse = len(map.layout) - 1 - y

graph.add\_node((x, y\_reverse))

# Add the edges (connections) to the graph

for node in graph.nodes:

x, y = node

# Check the top cell (y + 1)

if (x, y + 1) in graph.nodes:

graph.add\_edge(node, (x, y + 1))

# Check the bottom cell (y - 1)

if (x, y - 1) in graph.nodes:

graph.add\_edge(node, (x, y - 1))

# Check the left cell (x - 1)

if (x - 1, y) in graph.nodes:

graph.add\_edge(node, (x - 1, y))

# Check the right cell (x + 1)

if (x + 1, y) in graph.nodes:

graph.add\_edge(node, (x + 1, y))

return graph

# reads the text file and returns the content as a string

def readMapFile(filename, multiple\_endpoints=False):

'''

This function reads the map file and returns the content as a string.

Parameters:

filename (str): name of the file

Returns:

content (str): content of the file

'''

# get the directory of the absolute path leading to the script it is running on (so it will be correct when another script imports this)

path = os.path.dirname(os.path.abspath(sys.argv[0]))

print(path)

# join the directory path with the filename

abs\_file\_path = os.path.join(path, filename)

# check if file exists

if os.path.exists(abs\_file\_path):

with open(abs\_file\_path) as f:

content = f.read()

content = content.splitlines() # split the content into lines

# array of possible characters in the map

possible\_chars = ['X', 's', 'e', '.']

# scan through the content and check if it is a valid character

# also checks if there is only one start and end point, and returns the coordinates of the start and end points

start\_point = None

end\_point = None

for y in range(len(content)):

# split the line into characters

content[y] = list(content[y])

for x in range(len(content[y])):

if content[y][x] not in possible\_chars:

print("Error: Invalid character in map")

return None, None, None

elif content[y][x] == 's':

if start\_point != None:

print("Error: Multiple start points in map")

return None, None, None

else:

reversed\_y = len(content) - 1 - y

start\_point = (x, reversed\_y)

elif content[y][x] == 'e':

if end\_point != None and multiple\_endpoints == False:

print("Error: Multiple end points in map")

return None, None, None

else:

reversed\_y = len(content) - 1 - y

end\_point = (x, reversed\_y)

return content, start\_point, end\_point

else:

print("Error: File does not exist")

return None, None, None

import turtle

# program logic imports

from classes.utils import Utils

from classes.map import Map

from classes.drone import Drone

from classes.dronecontroller import DroneController

from classes.objectstatemanager import ObjectStateManager

from classes.terrainchanger import TerrainChanger

# pathfinder imports

from classes.lefthandpathfinder import LeftHandPathfinder

from classes.shortestpathfinder import ShortestPathfinder

from classes.travellingsalesmanpathfinder import TravellingSalesmanPathfinder

# graphics related imports

from classes.sprite import Sprite

from classes.renderer import Renderer

# user interaction imports

from classes.gui import GUI

# class to control the logic of the program

class Application:

"""

This class contols the logic of the program

Parameters:

program\_name (str):

authors (str):

class\_name (str):

Attributes:

program\_name (str):

authors (str):

class\_name (str):

title (str):

"""

# metadata for the program

def \_\_init\_\_(self, program\_name, authors, class\_name):

self.program\_name = program\_name

self.authors = authors

self.class\_name = class\_name

self.title = f"{self.program\_name}: Done by {self.authors} {self.class\_name}"

def startProgram(self):

"""

This the main method of the main program and runs the program logic.

Parameters:

file\_name (str): name of the map file to be read

Returns:

None

"""

# create turtle screen and display title

window = turtle.Screen()

### -------------------------------------------------------

### GUI CLASS: CAPTURES KEY PRESS EVENTS

### -------------------------------------------------------

gui = GUI(window)

filename = gui.get\_input("Map File Input", "Please enter file name:")

### -------------------------------------------------------

### READ AND SCAN MAP FILE

### -------------------------------------------------------

print("Reading and scanning map file")

map\_text, start\_pos, end\_pos = Utils.readMapFile("map\_files/" + filename)

if map\_text == None:

return

### -------------------------------------------------------

### RUN ALL PROGRAM LOGIC FIRST BEFORE RENDERING ANYTHING

### -------------------------------------------------------

# instantiate map and drone objects

map = Map(map\_text, start\_pos, end\_pos)

drone\_name = "drone1"

drone = Drone(drone\_name, map.start\_pos)

drone\_controller = DroneController(drone)

# get graph from map

map\_graph = Utils.map\_to\_graph(map)

# instantiate terrain changer object

terrain\_changer = TerrainChanger(map, map\_graph)

# instantiate lefthand and shortest pathfinder objects

pathfinders = ObjectStateManager([LeftHandPathfinder(), ShortestPathfinder()])

# tsp pathfinder is initialized outside the object state manager, as it is not required to be cycled through

tsp\_pathfinder = TravellingSalesmanPathfinder()

solution = pathfinders.current.solve(map\_graph)

# set instructions for drone controller

drone\_controller.instructions = solution

### -------------------------------------------------------

### LOADING NECCESSARY GRAPHICS ASSETS

### -------------------------------------------------------

renderer = Renderer(pixel\_size=25)

pixel\_ratio = renderer.pixel\_size / 20 # 20 pixels is 1 unit for shapesize()

# load all required graphic objects into renderer

renderer.add\_sprite(Sprite(name=drone\_name, color="red", current\_pos=drone.current\_pos, orientation=drone.orientation))

renderer.add\_sprite(Sprite(name="writer", color="black", pencolor="black", speed=0))

renderer.add\_sprite(Sprite(name="wall", color="grey", pencolor="black", shape="square", shapesize=pixel\_ratio, speed=0))

renderer.add\_sprite(Sprite(name="road", color="white", pencolor="black", shape="square", shapesize=pixel\_ratio, speed=0))

renderer.add\_sprite(Sprite(name="startpoint", color="#61ff6e", pencolor="black", shape="square", shapesize=pixel\_ratio, speed=0))

renderer.add\_sprite(Sprite(name="endpoint", color="#56defc", pencolor="black", shape="square", shapesize=pixel\_ratio, speed=0))

renderer.add\_sprite(Sprite(name="guidelight", color="yellow", pencolor="black", shape="circle", shapesize=pixel\_ratio, speed=0))

### -------------------------------------------------------

### RENDERING GRAPHICS

### -------------------------------------------------------

title\_bar = f"{pathfinders.current.algorithm\_name}, Steps Taken: {drone.steps\_taken}"

window.title(title\_bar)

# use Renderer to render map, title and spawn the drone

renderer.render\_map(map)

renderer.render\_title(self.title)

renderer.render\_drone(drone, spawn=True)

renderer.render\_dronetrail(drone.name)

renderer.render\_no\_solution(solution)

### -------------------------------------------------------

### CATCH KEY PRESS EVENTS, THIS IS THE "MAIN LOOP" OF THE PROGRAM

### -------------------------------------------------------

### series of commands to be executed on key press

# this activates on "m" key press, it moves the drone, renders new position and updates the title bar

gui.move\_drone\_event\_listener(drone\_controller, renderer, drone, pathfinders, terrain\_changer, map\_graph)

# this activates on "Tab" key press, it changes the pathfinder algorithm and updates the title bar

gui.change\_algorithm\_event\_listener(drone\_controller, renderer, drone, pathfinders, map\_graph)

# this activates the TSP algorithm, once this is activated, Tab out is disabled

gui.tsp\_event\_listener(drone\_controller, renderer, drone, tsp\_pathfinder)

# listen for key presses

window.onkey(

gui.command\_move\_drone, "m"

)

window.onkeypress(

gui.command\_move\_drone, "n"

)

window.onkey(

gui.command\_change\_algorithm, "Tab"

)

# Extra feature activation keys (Jun Jie)

window.onkey(

gui.command\_activate\_tsp, "1"

)

window.onkey(

gui.command\_activate\_random\_obstacles, "2"

)

# Extra feature activation keys (Tim)

window.onkey(

gui.command\_activate\_random\_maze, "3"

)

# window.onkey(

# gui.2nd\_feature, "4"

# )

window.listen()

### this must be the last line in the turtle program

window.mainloop()