Project Report DSAA CA2

Group 10

Khor Yu Yang, Nevan Ang Kai Wen

DAAA/FT/2B/01

# **Description/User-Guidelines**

The main idea behind the Pizza Runner project is to showcase our two algorithms, left-hand algorithm and the Breadth First Search algorithm, both of which are path finding algorithms. In this context, it is used to find the pizza delivery location, colourised as blue and the start location,colourised as green in the below example when starting the program.

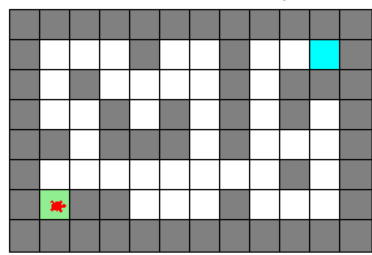


Figure 1 Picture of a sample map

To begin our application, type in cmd python <pizzarunner.py> <city map text file>



Figure 2 Sample of what to type

Users should also not include any other characters other than X(walls),e(pizza delivery location),’.’(road) and ‘s’(start location of drone) in the text file

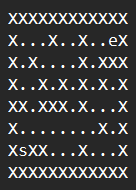


Figure 3 Sample of text file

To start off, by pressing the ‘Tab’ key, the maze will generate an automated path from the starting point to the ending point. The program consists of two algorithms to help track the correct path from the start to the end. Pressing the ‘Tab’ key allows the user to switch the algorithm multiple times. The program also tracks the time and amount of steps the turtle has taken as the program runs. Once an algorithm is selected,pressing ‘Space’ allows the user to begin the running of the program.

Wall Follower Algorithm (Left Hand Rule)

In our turtle program, we applied the wall follower algorithm using the left hand rule, which is based on the simple logic of movements whereby it follows along the left side of the enclosure wall all the way to the exit. It is a widely used path finding method and algorithm. Therefore, we applied this algorithm into our program to let the turtle find the exit.

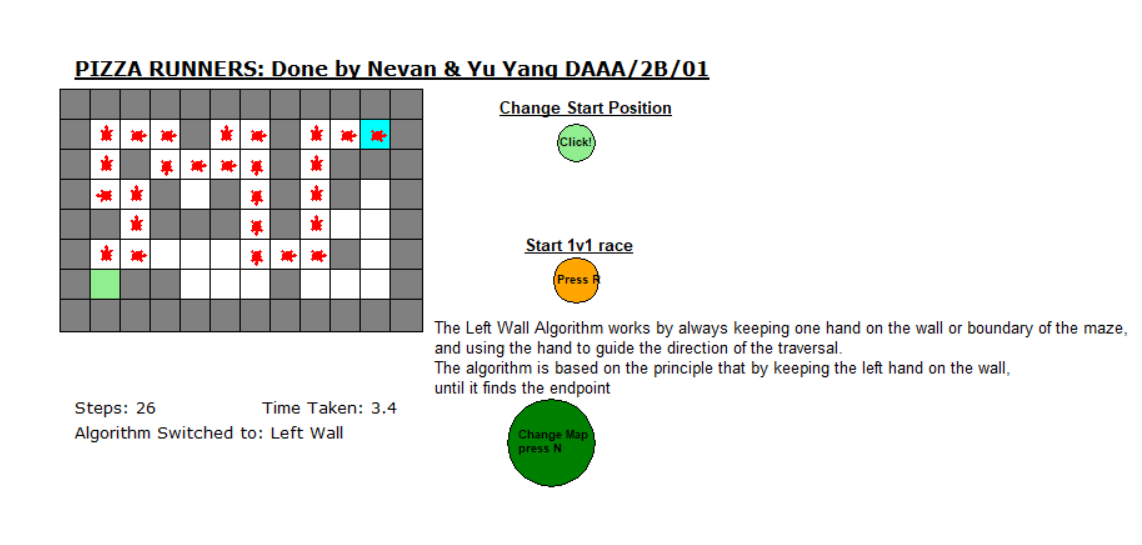


Figure 4 Turtle path using wall follower algorithm (left-hand rule)

Above picture provides a sample on how the algorithm works in a sample map from its starting point(green) to the final point(cyan). We can see that the direction of the turtle is constantly checking whether there is a left wall, and turning itself to that direction and moving to the next path. The big O notation for the left hand rule algorithm is O(n). However, there are limitations to this algorithm that it will face. Other than the amount of time it takes to find itself to the end point is longer than the other algorithms such as Breadth First Search, it might also enter an infinite loop when there are multiple paths that lead to different destinations. Below shows an example of an infinite loop

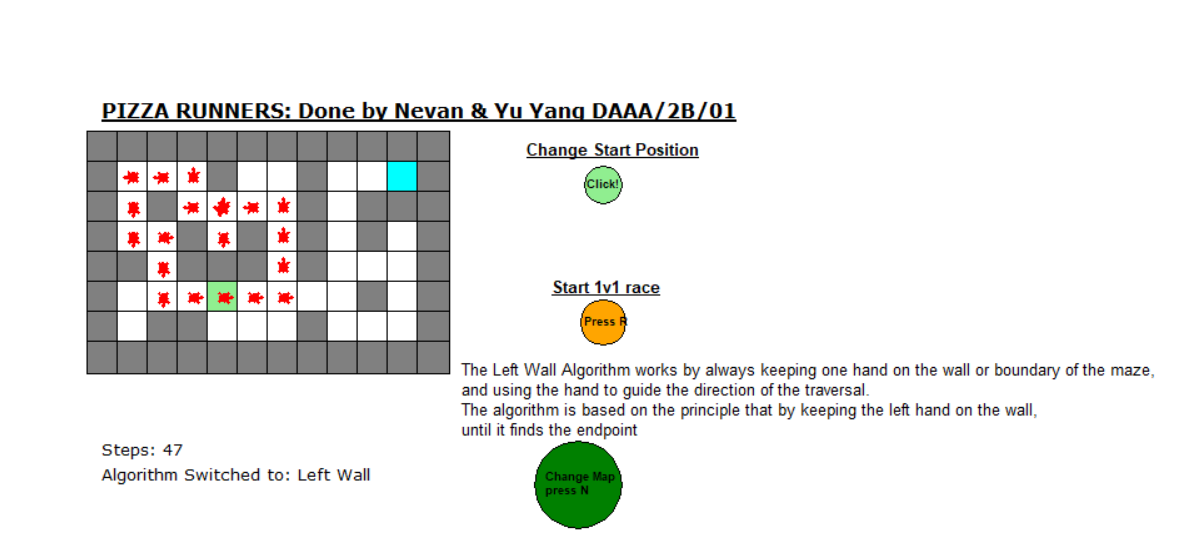


Figure 5 Infinite Loop(left-hand rule)

Figure 5 shows that the left wall follower enters a infinite loop when there is multiple paths that could possibly lead to endpoint, and since it follows the left walls, it is stuck in the loop. Therefore, left-wall algorithm does not always end up in an solution, which is what the company’s drones faced, so we need to implement other algorithms and see if it would helps us solves this issue and even find a shorter path to endpoint.

Breadth First Search Algorithm

Breadth First Search is a tree traversal and search algorithm which begins at a tree root, explores its neighbours or nodes that are nearest to it, before moving to its subsequent neighbouring node and exploring its neighbours. This is called depth, where there are layers of depth based on the distance a node is from the tree root.

In the case of our Pizza Runner, the tree root would be the start square block and the nodes are the road square blocks.

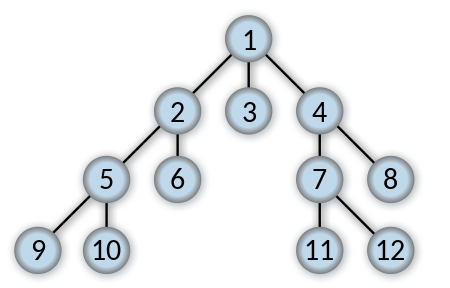


Figure 6From *Wikipedia,* Traversal order of a Breadth First Search Algorithm

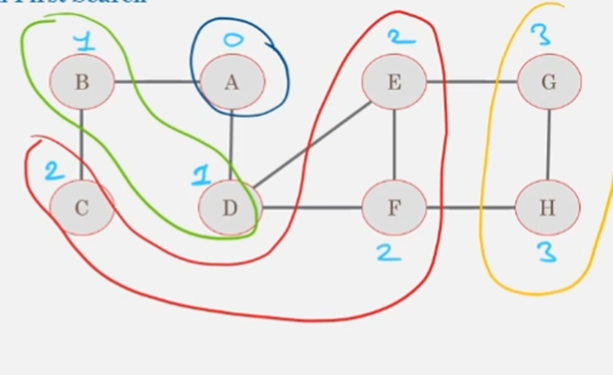


Figure 7 Depth visualised on a graph

Using figure 6 as an example, layers of depth are denoted by the different colours, the blue circle being the parent node and the first layer (0), the neighbours of the parent node being the second layer (1) and so on. The traversal order shows how the nodes are layered as well, the first layer being the nearest to the parent node.

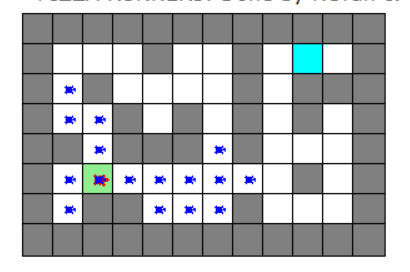


Figure 8 How the algorithm branches out to explore all paths

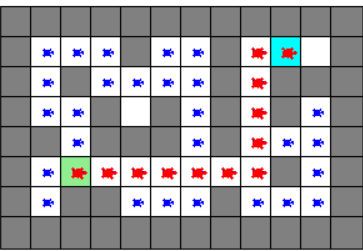


Figure 9 Finding the optimal path

Breadth First Search uses a queue data structure to keep track of the current node to find the node’s neighbours, the node(s) closest to it and append them inside. It is also used to keep track of nodes that have been found but not yet explored, in other words the said node’s neighbours' neighbours have not yet been found. Once the parent node’s neighbours have been found, it will be dequeued which is a list equivalent of popleft. This follows the queue’s First In First Out method. The cycle repeats until all the nodes have been explored.

Similar to what we have written above, we apply the same method to our pizza runner programme. We also add a visited list to keep track of all nodes that have been visited so as to not generate any duplicates and a solution dictionary which is essentially where we store all of the paths and node connections.

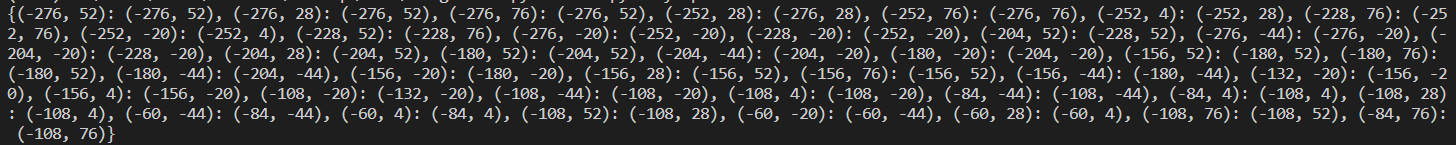


Figure 10 Example of solution dictionary

Once all the connections of the roads are found, how it finds a path is by recursive backtracking, from the end square cell coordinates to its neighbour, its neighbours’ neighbours and so on until it reaches the parent node, the start square cell coordinates. The Big O-Notation for Breadth First Search is O(n), where n represents the number of nodes in the graph. This means that the algorithm's time complexity increases linearly with the number of nodes in the graph.

Hence the limitation or problem when it comes to breadth first search is that although it will find the best path, it will take a long time if the map is large.

Basic Features

In order to get both algorithms mentioned to implement on the turtle program for comparison, we decided to create a method that could help us toggle between the two algorithms and see each of their effects on the program. It starts off with the left hand algorithm, pressing ‘Tab’ enables u to switch to Breadth First Search and back to the left hand algorithm. The title screen of the turtle shows the algorithm being used and how many steps it takes to reach the end location. Whenever an algorithm is run, the time it takes will also appear.

The titlebar also shows the current algorithm being used and its total steps once done, if path is not found, a prompt stating that will appear.It will also tell u the current algorithm that u are on based on a GUI prompt.

Applying Object Oriented Programming (OOP)

Inside our main program, we used a queue data structure for the Breadth First Search algorithm. It is used to store the next set of nodes or cells that need to be visited, like having a temporary memory.The queue also acts as a storage area for the next set of nodes to be visited, making it easy for the algorithm to continue the search process without having to go back to the start node. Overall, the use of the queue data structure in the BFS algorithm helps to ensure that the algorithm operates efficiently and effectively in finding the goal node.

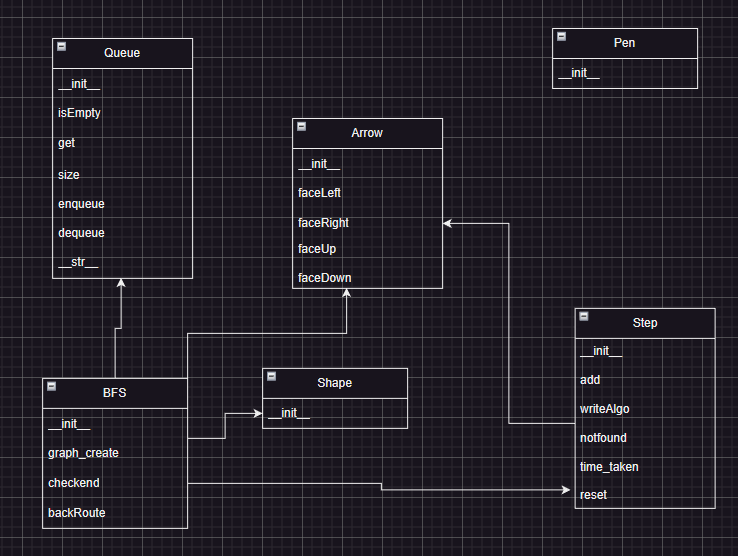


Figure 11 Relation between classes

The relation to shape, step and arrow is for the user interface, showing the total steps count done by Step, showing the route exploration and best route backtracking found in Shape and Arrow respectively.

Challenges Faced

A challenge we faced mainly was understanding the Turtle graphics and screen function in turtle. Creating the pens to draw the maze, mapping the paths was a task we found to be challenging. Our solution was to create multiple pens to perform different features. One for creating the maze, one for creating the title, one for mapping out the paths on the map, a pen to calculate steps, two separate pens for the left hand and Breadth First Search algorithm mapping out and exploring the maps. This resulted in quite a number of classes due to a lot of pens. The minimal experience with turtle affected the efficiency of the program.

Key Takeaways

The key takeaways of this project was greatly improving our knowledge and understanding in graph traversal and maze solving algorithms. How nodes and its connections enable a program to backtrack its connections to its original node and find a solution. Another takeaway was also understanding turtle graphics, how to create boundaries based off a certain symbol in a text file (‘X’) and create roads for a turtle sprite to move. Its graphics library, although sometimes hard to understand, allowed us to reinforce basic programming concepts, such as loops, functions, and conditionals.

Contribution

|  |  |
| --- | --- |
| BFS.py | Nevan Ang Kai Wen |
| Queue.py | Nevan Ang Kai Wen |
| shape.py | Nevan Ang Kai Wen |
| step.py | Nevan Ang Kai Wen |
| pen.py | Khor Yu Yang |
| main.py | Khor Yu Yang |
| arrow.py | Khor Yu Yang |

The maze creation,left hand algorithm, ‘Tab’ and ‘Space’ switching and activating algorithms is done by Yu Yang

Breadth FIrst Search, graph traversal exploration GUI and step tracking is done by Nevan

References

Wikipedia, 2023 [image] *Breadth-first search* Available from: [Breadth-first search - Wikipedia](https://en.wikipedia.org/wiki/Breadth-first_search)

[Accessed 8/2/2023]ace.py

Appendix

#Nevan’s Code

dfs.py

import turtle

# Depth First Search Algorithm

class DFS(turtle.Turtle):

def \_\_init\_\_(self,stack,road,start,end,Step,Shape,Arrow):

turtle.Turtle.\_\_init\_\_(self)

self.penup()

self.hideturtle()

self.shape("turtle")

self.shapesize(0.5, 0.5, 1)

self.color("red")

self.next\_cell=stack #list of cells to go to next

self.road=road #positions in turtle(x,y)

self.start=start

self.Step = Step

self.shape = Shape

self.arrow = Arrow

self.end=end

self.visited=set()#set

self.solution={}

#Road square block is added to a stack and the function continues to visit the neighbors of the Road square block until a it has no more neighbors to visit.

#Once all neighbors of that certain block have been visited, the function backtracks to the previous node by popping the last node

# from the stack and visiting its neighbors. This continues until the goal is found or there are no more nodes to visit.

def graph\_create(self):

x,y=self.start[0],self.start[1]

self.next\_cell.push((x, y)) # add the x and y position to the stack, meant to

self.solution[x,y] = x,y # add x and y to the solution dictionary

while self.next\_cell.size() > 0: # loop until the frontier list is empty

current=(x,y)

if(x - 24, y) in self.road and (x - 24, y) not in self.visited: # check left

cellleft = (x - 24, y)

self.solution[cellleft] = x, y # backtracking routine [cell] is the previous cell. x, y is the current cell

self.shape.goto(cellleft)

self.shape.stamp()

self.next\_cell.push(cellleft) # add cellleft to the cell next list

if (x, y - 24) in self.road and (x, y - 24) not in self.visited: # check down

celldown = (x, y - 24)

self.solution[celldown] = x, y

self.shape.goto(celldown)

self.shape.stamp()

self.next\_cell.push(celldown)

if(x + 24, y) in self.road and (x + 24, y) not in self.visited: # check right

cellright = (x + 24, y)

self.solution[cellright] = x, y

self.shape.goto(cellright)

self.shape.stamp()

self.next\_cell.push(cellright)

if(x, y + 24) in self.road and (x, y + 24) not in self.visited: # check up

cellup = (x, y + 24)

self.solution[cellup] = x, y

self.shape.goto(cellup)

self.shape.stamp()

self.next\_cell.push(cellup)

#current cell becomes last entry in frontier list

x, y = self.next\_cell.pop()

# print(x)

self.visited.add(current) # add current cell to visited list

self.shape.goto(x,y) # green turtle goto x and y position

self.shape.stamp() # stamp a copy of the green turtle on the maze

def backRoute(self,end\_x,end\_y):

self.arrow.goto(end\_x, end\_y)

self.arrow.stamp()

if ((end\_x,end\_y)) not in self.solution:

self.Step.notfound()

#if end coord not in dictionary

else:

if (end\_x, end\_y) == (self.start[0], self.start[1]):

return

else:

self.Step.add()

next\_x, next\_y = self.solution[end\_x, end\_y]

self.backRoute(next\_x, next\_y)

maploader.py

import turtle, os

class MapLoader(turtle.Turtle):

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

turtle.Turtle.\_\_init\_\_(self)

self.hideturtle()

self.shapesize(3.5,3.5)

self.shape("circle")

self.color("black", "green")

self.penup()

self.goto(80, -170)

self.stamp()

self.goto(55, -180)

self.write('Change Map\npress N',font=("Arial", 7, "bold"))

def map\_loader(self, pen,screen):

filename=screen.textinput("Map Changer", "Enter name of file")

if filename!=None and filename!='':

if os.path.exists(filename)==False:

pen.goto(55, -200)

pen.write('Map not found',font=("Arial", 10, "bold"))

else:

pen.clear()

pen.hideturtle()

pen.shape("square")

pen.shapesize(20/15)

pen.color("black",'grey')

pen.penup()

pen.goto(-300,120)

pen.write('PIZZA RUNNERS: Done by Nevan & Yu Yang DAAA/2B/01',font=("Verdana",12, "normal"))

with open(filename) as f:

lines = f.readlines()

return lines

else:

return

stack.py

class Stack():

def \_\_init\_\_(self): #init empty list

self.\_\_list=[]

def isEmpty(self):

return self.\_\_list == []

def size(self):

return len(self.\_\_list)

def clear(self):

self.\_\_list.clear()

def pop(self):

if self.isEmpty():

return None

else:

return self.\_\_list.pop()

def push(self, item):

self.\_\_list.append(item)

def get(self):

if self.isEmpty():

return None

else:

return self.\_\_list[-1]

def \_\_str\_\_(self):

output = '<'

for i in range( len(self.\_\_list) ):

item = self.\_\_list[i]

if i < len(self.\_\_list)-1 :

output += f'{str(item)}, '

else:

output += f'{str(item)}'

output += '>'

return output

Yu Yang’s Code

Changestart.py

import turtle, math

class SetStart(turtle.Turtle):

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

turtle.Turtle.\_\_init\_\_(self)

self.flag = True

self.start = start

self.end = end

self.walls = walls

self.penup()

self.hideturtle()

self.shape("turtle")

self.shapesize(0.5, 0.5, 1)

self.color("red")

self.goto(start)

self.showturtle()

# self.stamp()

self.speed(2)

def move\_forward(self):

if self.flag:

self.flag = False

x, y = self.position()

# Calculate the new position based on the heading and distance

new\_x = round(x + 24 \* math.cos(math.radians(self.heading())), 0)

new\_y = round(y + 24 \* math.sin(math.radians(self.heading())), 0)

if (new\_x,new\_y) not in self.walls:

self.forward(24)

self.flag = True

def turn\_left(self):

if self.flag:

self.flag = False

self.left(90)

self.flag = True

def turn\_right(self):

if self.flag:

self.flag = False

self.right(90)

self.flag = True

def move\_back(self):

if self.flag:

self.flag = False

x, y = self.position()

# Calculate the new position based on the heading and distance

new\_x = round(x - 24 \* math.cos(math.radians(self.heading())), 0)

new\_y = round(y - 24 \* math.sin(math.radians(self.heading())), 0)

if (new\_x,new\_y) not in self.walls:

self.backward(24)

self.flag = True

def markLoc(self):

if self.flag:

self.flag = False

if self.pos() == self.end:

return

self.start = self.pos()

return self.start

Race.py

import turtle, math

from changestart import SetStart

class OneVsOne(SetStart):

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

super().\_\_init\_\_(start,end,walls)

self.flag = True

self.start = start

self.end = end

self.walls = walls

self.penup()

self.hideturtle()

self.shape("arrow")

self.shapesize(0.5, 0.5, 1)

self.color("orange")

self.goto(start)

self.showturtle()

self.speed(2)

def move\_forward(self):

super().move\_forward()

if self.pos() == self.end:

return True

def turn\_left(self):

super().turn\_left()

def turn\_right(self):

super().turn\_right()

def move\_back(self):

super().move\_back()

if self.pos() == self.end:

return True

Main.py Yu Yang’s functions

# control turtle sprite

def controlStart(x,y,but):

global flag\_but

screen.title('Pizza Delivery')

if flag\_but and flag and mapflag:

# inner function to mark the new start point

def newStart():

global flag\_but

nonlocal setS

# Disable onkeypress event handler

screen.onkeypress(None, "m")

pen.goto(start[0])

pen.color('black','white') # change the original start point to part of road

road.append(start[0])

pen.stamp() #draw the square

temp = setS.markLoc()

if temp is None: # when user set start at end

arrow.showturtle()

# reset flag

flag\_but = True

# reset button

but.clear()

start\_create\_button()

start[0] = round(temp[0],0), round(temp[1], 0)

setS.hideturtle()

arrow.goto(start[0]) # change the set point to start point

pen.goto(start[0])

pen.color('black','lightgreen')

pen.stamp() #draw the square

arrow.showturtle()

# reset flag

flag\_but = True

# reset button

but.clear()

start\_create\_button()

flag\_but = False

arrow.clear()

arrow.hideturtle()

shape.clear()

# enable turtle to show that button is unavailable now

but.color("black", "red")

but.goto(100, 70)

but.stamp()

but.goto(88, 65)

but.write('Wait!',font=("Arial", 7, "bold"))

setS = SetStart(start[0],end[0],walls) # let user control sprite

screen.onkeypress(setS.move\_forward,'w') # forward key

screen.onkeypress(setS.turn\_left,'a') # turn left key

screen.onkeypress(setS.move\_back,'s') # move backward key

screen.onkeypress(setS.turn\_right,'d') #turn right key

screen.onkeypress(newStart,'m') # mark current coord as start point

screen.listen() #wait for keyboard input

# Button Function

def start\_create\_button():

button = turtle.Turtle()

button.hideturtle()

button.shape("circle")

button.color("black", "lightgreen")

button.shapesize(1.5, 1.5)

button.penup()

button.goto(100, 70)

button.stamp()

button.goto(88, 65)

button.write('Click!',font=("Arial", 7, "bold"))

text = turtle.Turtle()

text.hideturtle()

text.color('black')

text.penup()

text.goto(40, 90)

# text.pendown()

text.write('Change Start Position', font=("Arial", 10, "bold", 'underline'))

# text.goto(100,0)

screen.onscreenclick(lambda x, y: controlStart(x, y, button) if button.distance(x, y) < 30 else None)

# race counter

race\_counter = 1

# Race Function

def raceF(x,y,but,result):

# Flags to ensure no other functions interrupts it

global flag\_but

screen.title('Pizza Delivery')

if flag\_but and flag and mapflag:

# Inner Function to check if it is at the end point

def ifEndF():

global flag\_but, race\_counter

nonlocal result

# check if it reaches the end point

if one\_one.move\_forward():

print('Reached the end')

time2 = time.time() # get end time

time\_taken = round((time2 - time1), 2) # calculate time taken

result[user\_name] = time\_taken # store into hashtable

but.goto(-250,200)

but.write(f'{user\_name}, You took {time\_taken} seconds',font=("Arial", 12, "normal"))

if race\_counter != 2: # run function again for second player

race\_counter += 1 #add counter to track current player

one\_one.hideturtle()

one\_one.clear()

time.sleep(2) #delay

# undo two steps for button to revert for second player

but.undo()

but.undo()

but.undo()

flag\_but = True

raceF(x,y,but,result)

elif race\_counter == 2:

one\_one.hideturtle()

one\_one.clear()

time.sleep(2)

but.undo() #undo to last action

# check if both player result are the same

if result.buckets[0] == result.buckets[1]:

but.write(f'Wow! Both of you have the same result of {result.buckets[0]}. TIE',font=("Arial", 10, "normal"))

time.sleep(2)

but.clear()

flag\_but = True

arrow.showturtle()

race\_counter = 1 # reset race counter for next player

best = 0

race\_button() # rerun function to reset button

return

best = min(result.buckets) # find minimum time for best player

# Loop to reveal results of the race maze

if result[result.keys[0]] == best:

but.write(f'Congrats to {result.keys[0]}, with best timing of {result[result.keys[0]]} seconds',font=("Arial", 10, "normal"))

but.goto(-250,180)

but.write(f"Better Luck to {result.keys[1]}, with {result[result.keys[1]]} seconds",font=("Arial", 10, "normal"))

else:

but.write(f'Congrats to {result.keys[1]}, with best timing of {result[result.keys[1]]} seconds',font=("Arial", 10, "normal"))

but.goto(-250,180)

but.write(f"Better Luck to {result.keys[0]}, with {result[result.keys[0]]} seconds",font=("Arial", 10, "normal"))

time.sleep(2)

but.clear() #clear button

flag\_but = True

arrow.showturtle()

screen.onkeypress(None, 'w') # disable keypress

race\_counter = 1

best = 0

race\_button() #re run function to reset button

return

# Obtain user name for better tracking

user\_name = ""

while user\_name == '':

user\_name = screen.textinput("Input", "Enter your username:")

if not user\_name or race\_counter ==1 and user\_name =="":

return

flag\_but = False

# clear the turtle on the map for the race to occur

arrow.clear()

arrow.hideturtle()

shape.clear()

# algo\_step.clear()

# steps.clear()

# enable turtle to show that button is unavailable now

but.color("black", "red")

but.goto(100, -40)

but.stamp()

but.goto(88, -45)

but.write('Wait!',font=("Arial", 7, "bold"))

one\_one = OneVsOne(start[0],end[0],walls) # initialize class for race

# countdown timer before user can start controlling

but.goto(100,200)

for i in range(3, 0, -1):

but.write(f'Ready in {i}',font=("Arial", 10, "bold"))

time.sleep(1)

but.undo()

# but.undo()

but.write('GO!',font=("Arial", 10, "bold"))

time1 = time.time()

screen.onkeypress(ifEndF, 'w') # forward key

screen.onkeypress(one\_one.turn\_left,'a') # turn left key

# screen.onkeypress(ifEndB,'s') # move backward key

screen.onkeypress(one\_one.turn\_right,'d') #turn right key

screen.listen() #wait for keyboard input

# Button for OnevsOne competition

def race\_button():

# initialize hashtable

race\_history = HashTable(size=2)

button = turtle.Turtle()

button.hideturtle()

button.shape("circle")

button.color("black", "orange")

button.shapesize(1.8, 1.8)

button.penup()

button.goto(100, -40)

button.stamp()

button.goto(86, -45)

button.write('Press R',font=("Arial", 7, "bold"))

text = turtle.Turtle()

text.hideturtle()

text.color('black')

text.penup()

text.goto(60, -20)

# text.pendown()

text.write('Start 1v1 race', font=("Arial", 10, "bold", 'underline'))

# text.goto(100,0)

# screen.onscreenclick(lambda x, y: raceF(x, y, button) if button.distance(x, y) < 30 else None)

screen.onkeypress(functools.partial(raceF,x=0,y=0,but=button,result=race\_history),'r')

Hash.py

# hashtable to store the result of players

class HashTable:

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

self.size = size

self.keys = [None] \* self.size

self.buckets = [None] \* self.size

# A simple remainder method to convert key to index

def hashFunction(self, key ):

return len(key) % self.size

# Deal with collision resolution by means of

# linear probing with a 'plus 1' rehash

def rehashFunction(self, oldHash ):

return (len(oldHash) + 1) % self.size

def \_\_setitem\_\_(self, key, value):

# print(value)

index = self.hashFunction( key)

startIndex = index

while True:

# If bucket is empty then just use it

if self.buckets[index] == None:

self.buckets[index] = value

self.keys[index] = key

break

else: # If not empty and the same key then just overwrite

if self.keys[index] == key:

self.buckets[index] = value

break

else: # Look for another available bucket

index = self.rehashFunction(key)

# We must stop if no more buckets

if index == startIndex:

break

def \_\_getitem\_\_(self,key):

index = self.hashFunction(key)

startIndex = index

while True:

if self.keys [index] == key: # Will be mostly the case unless value

# had been previously rehashed at insertion

# time

return self.buckets[index]

else: # Value for the key is somewhere else

# (due to imperfect hash function)

index = self.rehashFunction(key)

if index == startIndex:

return None

Group’s code

shape.py

import turtle

#Breadth First Search

#color all roads as yellow dots

class Shape(turtle.Turtle):

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

turtle.Turtle.\_\_init\_\_(self)

self.shapesize(20\*0.5/15\*0.5)

self.hideturtle()

self.shape('turtle')

self.color("blue")

self.penup()

self.speed('fastest')

Step.py

import turtle

#Pen to make Step count

class Step(turtle.Turtle):

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

turtle.Turtle.\_\_init\_\_(self)

self.hideturtle()

self.penup()

self.step=0

self.goto(-300,-150)

self.write(f'Steps: {self.step}',font=("Verdana",10, "normal"))

# add steps for each algorithm

def add(self):

self.clear()

self.step+=1

self.goto(-300,-150)

self.speed('fastest')

self.write(f'Steps: {self.step}',font=("Verdana",10, "normal"))

# generate informattion about algorithm

def writeAlgo(self, algo):

self.clear()

self.goto(-300,-170)

self.speed('fastest')

self.write(f'Algorithm Switched to: {algo}',font=("Verdana",10, "normal"))

self.goto(-60,-150)

if algo == 'Breadth First Search':

# Breadth first search

self.write(

"""

Breadth First Search (BFS) is an algorithm used for traversing or searching

in a graph or tree data structure, where it visits all the vertices of a graph

or all the nodes of a tree at the same level before moving on to the next level

""",

font=('Arial', 10, 'normal')

)

elif algo == 'Left Wall':

# left hand

self.write("""

The Left Wall Algorithm works by always keeping one hand on the wall or boundary of the maze,

and using the hand to guide the direction of the traversal.

The algorithm is based on the principle that by keeping the left hand on the wall,

until it finds the endpoint

""", font=('Arial',10,'normal')

)

elif algo == 'Depth First Search':

self.write("""

Depth First Search Algorithm

""", font=('Arial',10,'normal')

)

# when endpoint blocked/no endpoint

def notfound(self,):

self.clear()

self.goto(-50,-200)

self.speed('fastest')

self.write(f'Algorithm is unable to find a path',font=("Verdana",10, "normal"))

# Record time taken for each algorithm to complete the maze

def time\_taken(self, starttime, endtime):

self.goto(-150, -150)

self.speed('fastest')

self.write(f'Time Taken: {round((endtime-starttime),2)}', font=("Verdana", 10, "normal"))

# clear steps

def reset(self):

self.step=0

self.clear()

Queue.py

class Queue():

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

self.items=[]

def isEmpty(self):

return self.items == []

def get(self):

return self.items[0]

def size(self):

return len(self.items)

def enqueue(self, item):

self.items.append(item)

def dequeue(self):

if self.isEmpty():

return None

else: return self.items.pop(0)

def \_\_str\_\_(self):

output = '<'

for i in range( len(self.items) ):

item = self.items[i]

if i < len(self.items)-1 :

output += f'{str(item)}, '

else:

output += f'{str(item)}'

output += '>'

return output

Pen.py

import turtle

#Turtle Class

class Pen(turtle.Turtle):

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

turtle.Turtle.\_\_init\_\_(self)

self.shape("square")

self.shapesize(20/15)

self.color("black",'grey')

self.penup()

self.goto(-300,120)

self.write('PIZZA RUNNERS: Done by Nevan & Yu Yang DAAA/2B/01',

font=("Verdana",12, "bold", 'underline'))

self.speed(10)

Main.py

# import networkx

import turtle, sys, time, functools, os

from changestart import SetStart

from race import OneVsOne

from hash import HashTable

from step import Step

from shape import Shape

from bfs import BFS

from pen import Pen

from arrow import Arrow

from queue import Queue

from dfs import DFS

from stack import Stack

from maploader import MapLoader

#Create the canvas

screen=turtle.Screen()

screen.title('Pizza Delivery')

screen.bgcolor('white')

screen.setup(width=700,height=500)

#MAP setup function

def setup\_maze(level,road,walls,end,start):

for y in range(len(level)):

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

character = level[y][x]

screen\_x = -300 + (x \*24)

screen\_y = 100 - (y \* 24)

if character =="X": #Buildings

pen.goto(screen\_x, screen\_y)

pen.color('black','grey')

pen.stamp() #draw the square

walls.append((screen\_x,screen\_y))

elif character == ".": #Road

pen.goto(screen\_x,screen\_y)

road.append(pen.pos()) #append the position to list

pen.color('black','white')

pen.stamp() #draw the square

elif character == "s": #Start Point

pen.goto(screen\_x,screen\_y)

# arrow.goto(screen\_x,screen\_y)

start.append(pen.pos())

pen.color('black','lightgreen')

pen.stamp() #draw the square

elif character == "e": #End Point

pen.goto(screen\_x,screen\_y)

end.append(pen.pos())

road.append(pen.pos())#for bfs

pen.color('black','cyan')

pen.stamp() #draw the square

active\_func = 'left' #global variable

# Left Wall follower function

def left\_path():

while True: # Loop the action movement until it finds the end point and return False

if arrow.faceLeft()== False or arrow.faceRight()==False \

or arrow.faceUp()==False or arrow.faceDown()==False:

return

# toggle between three algorithms using tab

def toggle\_function():

global active\_func

if active\_func =='left':

active\_func = 'short'

algo\_step.writeAlgo('Breadth First Search')

elif active\_func == 'dfs':

active\_func = 'left'

algo\_step.writeAlgo('Left Wall')

elif active\_func == 'short':

active\_func = 'dfs'

algo\_step.writeAlgo('Depth First Search')

flag = True #global flag

# Run the algorithms with the toggled algorithm

def run\_algorithm():

global active\_func, flag, flag\_but

if flag and flag\_but and mapflag:

flag = False

arrow.clear()

shape.clear()

arrow.hideturtle()

arrow.goto(start[0])

arrow.showturtle()

steps.reset()

if active\_func == 'left': # Left hand Algorithm

starttime = time.perf\_counter()

left\_path()

endtime = time.perf\_counter()

steps.time\_taken(starttime, endtime) # calculate timetaken

screen.title(f"Pizza Delivery Current Algorithm: Left Hand, Steps Taken: {arrow.Step.step}")

flag = True

elif active\_func == 'short' : # Breadth First Search Algorithm

starttime = time.perf\_counter()

bfs=BFS(queue,road,start[0],end[0],steps,arrow,shape)

bfs.graph\_create()

bfs.backRoute(end[0][0],end[0][1])

endtime = time.perf\_counter()

steps.time\_taken(starttime, endtime) # calculate timetaken

screen.title(f"Pizza Delivery Current Algorithm: Breadth First Search, Steps Taken: {bfs.Step.step}")

flag = True

elif active\_func == 'dfs': # Depth First Search Algorithm

starttime = time.perf\_counter() # track start time

dfs=DFS(stack,road,start[0],end[0],steps,shape,arrow)

dfs.graph\_create()

dfs.backRoute(end[0][0],end[0][1])

endtime = time.perf\_counter()

steps.time\_taken(starttime, endtime) # calculate timetaken

screen.title(f"Pizza Delivery Current Algorithm: Depth First Search, Steps Taken: {dfs.Step.step}")

flag = True

flag\_but = True

mapflag=True

# control turtle sprite

def controlStart(x,y,but):

global flag\_but

screen.title('Pizza Delivery')

if flag\_but and flag and mapflag:

# inner function to mark the new start point

def newStart():

global flag\_but

nonlocal setS

# Disable onkeypress event handler

screen.onkeypress(None, "m")

pen.goto(start[0])

pen.color('black','white') # change the original start point to part of road

road.append(start[0])

pen.stamp() #draw the square

temp = setS.markLoc()

if temp is None: # when user set start at end

arrow.showturtle()

# reset flag

flag\_but = True

# reset button

but.clear()

start\_create\_button()

start[0] = round(temp[0],0), round(temp[1], 0)

setS.hideturtle()

arrow.goto(start[0]) # change the set point to start point

pen.goto(start[0])

pen.color('black','lightgreen')

pen.stamp() #draw the square

arrow.showturtle()

# reset flag

flag\_but = True

# reset button

but.clear()

start\_create\_button()

flag\_but = False

arrow.clear()

arrow.hideturtle()

shape.clear()

# enable turtle to show that button is unavailable now

but.color("black", "red")

but.goto(100, 70)

but.stamp()

but.goto(88, 65)

but.write('Wait!',font=("Arial", 7, "bold"))

setS = SetStart(start[0],end[0],walls) # let user control sprite

screen.onkeypress(setS.move\_forward,'w') # forward key

screen.onkeypress(setS.turn\_left,'a') # turn left key

screen.onkeypress(setS.move\_back,'s') # move backward key

screen.onkeypress(setS.turn\_right,'d') #turn right key

screen.onkeypress(newStart,'m') # mark current coord as start point

screen.listen() #wait for keyboard input

# Button Function

def start\_create\_button():

button = turtle.Turtle()

button.hideturtle()

button.shape("circle")

button.color("black", "lightgreen")

button.shapesize(1.5, 1.5)

button.penup()

button.goto(100, 70)

button.stamp()

button.goto(88, 65)

button.write('Click!',font=("Arial", 7, "bold"))

text = turtle.Turtle()

text.hideturtle()

text.color('black')

text.penup()

text.goto(40, 90)

# text.pendown()

text.write('Change Start Position', font=("Arial", 10, "bold", 'underline'))

# text.goto(100,0)

screen.onscreenclick(lambda x, y: controlStart(x, y, button) if button.distance(x, y) < 30 else None)

# race counter

race\_counter = 1

# Race Function

def raceF(x,y,but,result):

# Flags to ensure no other functions interrupts it

global flag\_but

screen.title('Pizza Delivery')

if flag\_but and flag and mapflag:

# Inner Function to check if it is at the end point

def ifEndF():

global flag\_but, race\_counter

nonlocal result

# check if it reaches the end point

if one\_one.move\_forward():

print('Reached the end')

time2 = time.time() # get end time

time\_taken = round((time2 - time1), 2) # calculate time taken

result[user\_name] = time\_taken # store into hashtable

but.goto(-250,200)

but.write(f'{user\_name}, You took {time\_taken} seconds',font=("Arial", 12, "normal"))

if race\_counter != 2: # run function again for second player

race\_counter += 1 #add counter to track current player

one\_one.hideturtle()

one\_one.clear()

time.sleep(2) #delay

# undo two steps for button to revert for second player

but.undo()

but.undo()

but.undo()

flag\_but = True

raceF(x,y,but,result)

elif race\_counter == 2:

one\_one.hideturtle()

one\_one.clear()

time.sleep(2)

but.undo() #undo to last action

# check if both player result are the same

if result.buckets[0] == result.buckets[1]:

but.write(f'Wow! Both of you have the same result of {result.buckets[0]}. TIE',font=("Arial", 10, "normal"))

time.sleep(2)

but.clear()

flag\_but = True

arrow.showturtle()

race\_counter = 1 # reset race counter for next player

best = 0

race\_button() # rerun function to reset button

return

best = min(result.buckets) # find minimum time for best player

# Loop to reveal results of the race maze

if result[result.keys[0]] == best:

but.write(f'Congrats to {result.keys[0]}, with best timing of {result[result.keys[0]]} seconds',font=("Arial", 10, "normal"))

but.goto(-250,180)

but.write(f"Better Luck to {result.keys[1]}, with {result[result.keys[1]]} seconds",font=("Arial", 10, "normal"))

else:

but.write(f'Congrats to {result.keys[1]}, with best timing of {result[result.keys[1]]} seconds',font=("Arial", 10, "normal"))

but.goto(-250,180)

but.write(f"Better Luck to {result.keys[0]}, with {result[result.keys[0]]} seconds",font=("Arial", 10, "normal"))

time.sleep(2)

but.clear() #clear button

flag\_but = True

arrow.showturtle()

screen.onkeypress(None, 'w') # disable keypress

race\_counter = 1

best = 0

race\_button() #re run function to reset button

return

# Obtain user name for better tracking

user\_name = ""

while user\_name == '':

user\_name = screen.textinput("Input", "Enter your username:")

if not user\_name or race\_counter ==1 and user\_name =="":

return

flag\_but = False

# clear the turtle on the map for the race to occur

arrow.clear()

arrow.hideturtle()

shape.clear()

# algo\_step.clear()

# steps.clear()

# enable turtle to show that button is unavailable now

but.color("black", "red")

but.goto(100, -40)

but.stamp()

but.goto(88, -45)

but.write('Wait!',font=("Arial", 7, "bold"))

one\_one = OneVsOne(start[0],end[0],walls) # initialize class for race

# countdown timer before user can start controlling

but.goto(100,200)

for i in range(3, 0, -1):

but.write(f'Ready in {i}',font=("Arial", 10, "bold"))

time.sleep(1)

but.undo()

# but.undo()

but.write('GO!',font=("Arial", 10, "bold"))

time1 = time.time()

screen.onkeypress(ifEndF, 'w') # forward key

screen.onkeypress(one\_one.turn\_left,'a') # turn left key

# screen.onkeypress(ifEndB,'s') # move backward key

screen.onkeypress(one\_one.turn\_right,'d') #turn right key

screen.listen() #wait for keyboard input

# Button for OnevsOne competition

def race\_button():

# initialize hashtable

race\_history = HashTable(size=2)

button = turtle.Turtle()

button.hideturtle()

button.shape("circle")

button.color("black", "orange")

button.shapesize(1.8, 1.8)

button.penup()

button.goto(100, -40)

button.stamp()

button.goto(86, -45)

button.write('Press R',font=("Arial", 7, "bold"))

text = turtle.Turtle()

text.hideturtle()

text.color('black')

text.penup()

text.goto(60, -20)

# text.pendown()

text.write('Start 1v1 race', font=("Arial", 10, "bold", 'underline'))

# text.goto(100,0)

# screen.onscreenclick(lambda x, y: raceF(x, y, button) if button.distance(x, y) < 30 else None)

screen.onkeypress(functools.partial(raceF,x=0,y=0,but=button,result=race\_history),'r')

# function to change map and reload the coordinates of properties in maze

def ChangeMap():

global road, walls, end, start,arrow, mapflag

if flag and flag\_but and mapflag:

mapflag = False

lines = maploader.map\_loader(pen,screen)

road = []

walls = []

end = []

start = []

setup\_maze(lines,road,walls,end,start)

arrow.clear()

arrow.hideturtle()

arrow = Arrow(start[0],steps,end,walls)

mapflag = True

#Main Program

pen=Pen()

shape=Shape()

road=[]

walls=[]

end=[] # start and end is in tuple, [(x,y)]

start=[]

queue=Queue()

stack=Stack()

#retrieve txt filename from command line

filename = sys.argv[-1] #last argument

#check if the file opened is txt

if filename[-4:]!=".txt":

raise Exception('Sorry, the file type specified was not correct...Please Try Again')

try:

with open(filename) as f:

lines = f.readlines()

except FileNotFoundError: #if filename is not correct

print('filename specified does not exists...Please Try Again')

setup\_maze(lines,road,walls,end,start)

steps=Step()

algo\_step=Step()

algo\_step.writeAlgo('Left Wall')

arrow=Arrow(start[0],steps,end,walls)

maploader = MapLoader()

start\_create\_button() #Create button for switching start point

race\_button() #Create button for starting the race

#tab key bind

screen.listen() #wait for keyboard input

screen.onkeypress(toggle\_function,'Tab') #trigger switch function algo

screen.onkeypress(run\_algorithm,'space') #trigger run function run algo

screen.onkeypress(ChangeMap,'n') #map changer

while True:

screen.update()

Bfs.py

import turtle

# Breadth First Search Class

class BFS(turtle.Turtle):

def \_\_init\_\_(self,queue,road,start,end, Step,arrow,shape):

turtle.Turtle.\_\_init\_\_(self)

self.penup()

self.hideturtle()

self.shape("turtle")

self.shapesize(0.5, 0.5, 1)

self.color("red")

self.queue=queue #the graph

self.road=road #positions in turtle(x,y)

self.start=start

self.Step = Step

self.end=end

self.visited=set()#set

self.solution={}

self.arrow = arrow

self.shape = shape

def graph\_create(self): #start x start y

self.endfound=False

self.queue.enqueue((round(self.start[0],0),round(self.start[1],0)))

self.solution[round(self.start[0],0),round(self.start[1],0)]=round(self.start[0],0),round(self.start[1],0)

while self.queue.size()>0:

self.x,self.y=self.queue.get()

# print(f'{self.x} X IS HERE')

# print(f'{self.y} y is here')

self.queue.dequeue() # Pop first element of Queue, Get popped element And begin traversal of A adjacent vertex

if self.endfound==True:

return

if(self.x - 24, self.y) in self.road and (self.x - 24, self.y) not in self.visited: # check the cell on the left

cell = (self.x - 24, self.y)

self.solution[cell] = self.x, self.y # backtracking routine [cell] is the previous cell. x, y is the current cell

self.queue.enqueue(cell) # add cell to deque

self.visited.add((self.x-24, self.y)) # add cell to visited list

self.checkend((self.x-24, self.y))

if (self.x, self.y - 24) in self.road and (self.x, self.y - 24) not in self.visited: # check the cell down

cell = (self.x, self.y - 24)

self.solution[cell] = self.x, self.y

#blue.goto(cell)

#blue.stamp()

self.queue.enqueue(cell)

self.visited.add((self.x, self.y - 24))

self.checkend((self.x, self.y - 24))

# print(self.solution)

if(self.x + 24, self.y) in self.road and (self.x + 24, self.y) not in self.visited: # check the cell on the right

cell = (self.x + 24, self.y)

self.solution[cell] = self.x, self.y

#blue.goto(cell)

#blue.stamp()

self.queue.enqueue(cell)

self.visited.add((self.x +24, self.y))

self.checkend((self.x +24, self.y))

if(self.x, self.y + 24) in self.road and (self.x, self.y + 24) not in self.visited: # check the cell up

cell = (self.x, self.y + 24)

self.solution[cell] = self.x, self.y

#blue.goto(cell)

#blue.stamp()

self.queue.enqueue(cell)

self.visited.add((self.x, self.y + 24))

self.checkend((self.x, self.y + 24))

#need to add end inside

self.shape.goto(self.x,self.y)

self.shape.stamp()

def checkend(self,coordinates):

if coordinates==self.end:

self.endfound=True

def backRoute(self,end\_x,end\_y):

self.arrow.goto(end\_x, end\_y)

# print(self.solution)

self.arrow.stamp()

if ((end\_x,end\_y)) not in self.solution:

self.Step.notfound()

#if end coord not in dictionary

else:

if (end\_x, end\_y) == (self.start[0], self.start[1]):

return

else:

self.Step.add()

next\_x, next\_y = self.solution[end\_x, end\_y]

self.backRoute(next\_x, next\_y)