**Technical Solution**

**Technical Solution Cover Sheet**

The following identifies any key aspects/code of the program which can be located underneath.

|  |  |  |
| --- | --- | --- |
| Algorithm | Page Number | Description |
| Get Neighbours method of Grid Class | 95-108 | This method finds the adjacent nodes of the current node selected. Nodes are derived from using a graph data structure |
| BreakWall function | 111-123 | Uses list manipulation to disable walls on the class Grid. Edits the attributes of nodes that have been entered as parameters |
| Recursive Backtrack | 126-135 | The algorithm explained in the Design documentation, works by recursion and for loop to ensure that every node is explored. This is designed to create the maze. |
| Depth First Search | 138-148 | Unlike the Recursive Backtrack, this uses recursion to find the maximum number of nodes needed to reach the target node. This is designed to solve the maze. |
| Maze Generator | 151-160 | Initialises all the nodes and graph data structure for Solve game mode and creates the maze by using the recursive backtrack algorithm. |
| solvecreateCheckWall | 172-201 | Uses comprehensive list manipulation and lots of variables to find whether a wall is in-between a node and the node intended to travel to. |
| Difficulty Verification | 221-227 | A Verification to assess whether the result of the Depth first search algorithm exceeds 90% of the maze. |
| Main Game Loop (Solving) | 230-298 | Responsible for handling all major game events in the Solve game-mode, from keyboard inputs to tracking the time taken to solve the maze. |
| Maze Verification | 301-322 | Encapsulates the difficulty verification and Maze generator to ensure that the right maze is created. |
| Heuristic method of grid class | 438-443 | Euclidian distance is calculated for the heuristic for the A\* pathfinding algorithm. |
| Node Initialiser | 470-479, 45 | Initialises all the nodes for the Create game Mode. |
| Create Maze | 482-535, 45 | Handles the game loop for the user to create a maze |
| Add Start/End points. | 538-595, 46 | Handles the game loop for the user to add a start and end point to their newly created maze. Uses a unique list manipulation to control how this works. |
| A\* Pathfinding | 598-633, 47 | Discussed in the Design documentation, the A\* pathfinder takes various parameters and returns the shortest path in a graph abstract data structure, using the maze created by the user, the nodes initialized, and the heuristics calculated. Lots of processes in this algorithm make this is prominent algorithm of the project. |
| Computer Main Game Loop | 664-698, 48 | Responsible for outputting the A\* pathfinding algorithm and the accompanying game loop. |
| User Main Game Loop | 701-756, 49 | Responsible for game loop for solving the created maze. Uses very similar processes to that of the Main game loop for solving. |

Technical Solution: The Code

The entire technical solution, line by line. The following will not work without the proper .jpg files downloaded.

import pygame, sys, random

from pygame.locals import \*

pygame.init()

clock = pygame.time.Clock()

width = 750

height = 750

screen = pygame.display.set\_mode((width+250,height+2))

pygame.display.set\_caption('Maze Game')

BLACK = (0,0,0)

WHITE = (255,255,255)

GREEN = (0,255,0)

BLUE = (0,0,255)

RED = (255,0,0)

YELLOW = (255,255,0)

TEAL = (0,150,150)

font = pygame.font.Font(None, 50)

font\_hints = pygame.font.Font(None, 30)

font\_color = pygame.Color('springgreen')

font\_white = pygame.Color(WHITE)

font\_blue = pygame.Color(BLUE)

font\_red = pygame.Color(RED)

font\_yellow = pygame.Color(YELLOW)

solve\_title = pygame.image.load(r'Default\_Solve.jpg')

solve\_menu = pygame.image.load(r'Default\_Solve\_Menu.jpg')

create\_logo = pygame.image.load(r'Default\_Create.jpg')

create\_menu = pygame.image.load(r'Default\_Create\_Menu.jpg')

create\_decision = pygame.image.load(r'Default\_Create\_Decision.jpg')

create\_dTitle = pygame.image.load(r'Default\_Create\_Choose.jpg')

title = pygame.image.load(r'Default\_Logo.jpg')

menu = pygame.image.load(r'Default\_Menu.jpg')

global timer #used in solveMaze

global passed\_time #used in solveMaze

timer = 0

passed\_time = 0

def Main():

while True:

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == KEYDOWN:

if event.key == K\_ESCAPE:

sys.exit()

if event.key == K\_1:

SolvingInit()

if event.key == K\_2:

CreationInit()

screen.fill(BLACK)

screen.blit(title, (150,20))

screen.blit(menu, (100,330))

pygame.display.update()

#MAZE SOLVING CODE

class Block:#includes information for the player to navigate the grid

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def draw(self, colour):

pygame.draw.rect(screen, colour, (self.x+1, self.y+1, p\_width-2, p\_width-2))

player = Block(0,0)

class Grid: #one instance of this = one node

def \_\_init\_\_(self,x,y):

self.x = x

self.y = y

self.neighbours\_near = []

self.wall = [True, True, True, True]

self.is\_visited = False

def draw(self, colour): #draws individual grids

if self.wall[0]: #top

pygame.draw.line(screen, colour, (self.x\*p\_width, self.y\*p\_height), (self.x\*p\_width+p\_width, self.y\*p\_height), 3)

if self.wall[1]: #left

pygame.draw.line(screen, colour, (self.x\*p\_width, self.y\*p\_height+p\_height), (self.x\*p\_width, self.y\*p\_height), 3)

if self.wall[2]: #right

pygame.draw.line(screen, colour, (self.x\*p\_width+p\_width, self.y\*p\_height), (self.x\*p\_width+p\_width, self.y\*p\_height+p\_height), 3)

if self.wall[3]: #bottom

pygame.draw.line(screen, colour, (self.x\*p\_width+p\_width, self.y\*p\_height+p\_height), (self.x\*p\_width, self.y\*p\_height+p\_height), 3)

def get\_neighbours(self, node\_list):

for node in node\_list: #checking to see whether it has the same coordinates

if self.y > 0: #up node

if self.y - 1 == node.y and self.x == node.x:

self.neighbours\_near.append(node)

if self.x > 0: #left node

if self.y == node.y and self.x - 1 == node.x:

self.neighbours\_near.append(node)

if self.x < rows - 1: #right node

if self.y == node.y and self.x + 1 == node.x:

self.neighbours\_near.append(node)

if self.y < columns - 1: #down node

if self.y + 1 == node.y and self.x == node.x:

self.neighbours\_near.append(node)

def breakWall(node, neighbour):

if neighbour.x > node.x: #meaning is the neighbour on the right of node

node.wall[2] = False

neighbour.wall[1] = False #double break wall because there are 2 walls over each other covering different sides of the wall

if neighbour.x < node.x: #left wall

node.wall[1] = False

neighbour.wall[2] = False

if neighbour.y > node.y: #down wall

node.wall[3] = False

neighbour.wall[0] = False

if neighbour.y < node.y: #up wall

node.wall[0] = False

neighbour.wall[3] = False

def recursiveBacktrack(node, node\_list):

node.is\_visited = True

node.get\_neighbours(node\_list)

neighbours\_list = node.neighbours\_near

random.shuffle(neighbours\_list)

#neighbour list is now shuffled

for neighbour in neighbours\_list:

if neighbour.is\_visited == False:

breakWall(node, neighbour)

recursiveBacktrack(neighbour, node\_list) #recursion

def depthFirstSearch(start, end, node\_list, visited):

node = getNode(node\_list, int(start.x//p\_width), int(start.y//p\_width))

if node.x == end.x and node.y == end.y:

return True

visited.append(node)

neighbours = node.neighbours\_near

for n in neighbours:

if n not in visited and solvecreateCheckWall(start, node\_list, getDirection(node,n)) == False:

new = Block(n.x\*p\_width, n.y\*p\_width)

depthFirstSearch(new, end, node\_list, visited)

return visited

def mazeGenerator():

node\_list = []

for i in range(rows):

for j in range(columns):

grid = Grid(i, j)

node\_list.append(grid)

starting\_node = node\_list[0] #has (x,y) of (0,0)

recursiveBacktrack(starting\_node, node\_list)

return node\_list #put this in a for loop and .draw the objects will draw maze

def points(start\_x, start\_y, end\_x, end\_y): #start point and endpoint in one function

Block(start\_x, start\_y).draw(BLUE)

Block(end\_x, end\_y).draw(RED)

def markBlock(x, y, colour):

Block(x,y).draw(colour)

def solvecreateCheckWall(block, node\_list, direction): #only works if you place a Block as a parameter NOT a node

(x, y) = block.x, block.y

x, y = int(x//p\_width), int(y//p\_width) #this is a FLOAT cannot use this data type for anything so int() will convert to int! divides by pixel per square to find my standard coordinates not the pixel coordinates

if direction == 'r':

x += 1

check\_node = getNode(node\_list, x,y) #node\_list[20] for (1,0)

if check\_node.wall[1] == True:

return True

else:

return False

if direction == 'l':

check\_node = getNode(node\_list, x,y)

if check\_node.wall[1] == True:

return True

else:

return False

if direction == 'u':

check\_node = getNode(node\_list, x,y)

if check\_node.wall[0] == True:

return True

else:

return False

if direction == 'd':

y += 1

check\_node = getNode(node\_list, x,y)

if check\_node.wall[0] == True:

return True

else:

return False

def getNode(node\_list, x, y): #position of the node corresponding to the position of the block

for node in node\_list:

if (node.x, node.y) == (x, y):

return node

def getDirection(currentCell, n):

if currentCell.x > n.x:

return 'l'

if currentCell.x < n.x:

return 'r'

if currentCell.y > n.y:

return 'u'

if currentCell.y < n.y:

return 'd'

def difficultyVerification(path): #algorithm works by getting a path of the maximum nodes needed to traverse

numberOfnode = len(path)

percentage = (numberOfnode\*100)//(rows\*columns) #the maximum percentage of the maze that

if percentage >= 90:

return True

else:

return False

def solveMaze(pack, start\_time, passed\_time):

while True:

global timer

node\_list = pack[0]

start\_x = pack[1]

start\_y = pack[2]

end\_x = pack[3]

end\_y = pack[4]

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == KEYDOWN:

if event.key == K\_ESCAPE:

SolvingInit()

#player movement

if event.key == K\_RIGHT and player.x < width-p\_width:

if solvecreateCheckWall(player, node\_list, 'r') == False: #solvecreateCheckWall returning false means no wall

player.x += p\_width

if event.key == K\_LEFT and player.x > 0:

if solvecreateCheckWall(player, node\_list, 'l') == False:

player.x -= p\_width

if event.key == K\_DOWN and player.y < height-p\_width:

if solvecreateCheckWall(player, node\_list, 'd') == False:

player.y += p\_width

if event.key == K\_UP and player.y > 0:

if solvecreateCheckWall(player, node\_list, 'u') == False:

player.y -= p\_width

screen.fill(BLACK)

text1 = font\_hints.render('Use Directional Keys to', True, font\_white)

text2 = font\_hints.render('navigate the maze', True, font\_white)

text3 = font\_hints.render('Press ESC key to', True, font\_white)

text4 = font\_hints.render('Return to Menu', True, font\_white)

text5 = font\_hints.render('Get Stuck?', True, font\_yellow)

text6 = font\_hints.render('Follow the wall on the', True, font\_yellow)

text7 = font\_hints.render('left around the maze :)', True, font\_yellow)

screen.blit(text1, (width+20, 200))

screen.blit(text2, (width+20, 230))

screen.blit(text3, (width+20, 300))

screen.blit(text4, (width+20, 330))

screen.blit(text5, (width+20, 400))

screen.blit(text6, (width+20, 430))

screen.blit(text7, (width+20, 460))

points(start\_x, start\_y, end\_x, end\_y)

player.draw(GREEN)

for line in node\_list: #draws maze

line.draw(WHITE)

if int(player.x) == int(end\_x) and int(player.y) == int(end\_y):

return True

else:

passed\_time = pygame.time.get\_ticks() - start\_time #in milliseconds

timer = passed\_time

time = str(passed\_time/1000)

text = font.render(f'Time: {time}', True, font\_color)

screen.blit(text,(770, 100))

pygame.display.update()

def mazeVerification():

accepted = False

while accepted == False:

node\_list = mazeGenerator() #very important contains all information to draw the maze (loop for item in list and .draw)

#note: node\_list is a tree data structure, it contains no cycles!

end\_x = int(random.randint(rows-5,rows-1) \* p\_width)

end\_y = int(random.randint(columns-5,columns-1) \* p\_width)

start\_x = 0

start\_y = 0

endCell = getNode(node\_list, int(end\_x//p\_width), int(end\_y//p\_width))

start = Block(0,0)

visited = []

path = depthFirstSearch(start, endCell, node\_list, visited)

diff = difficultyVerification(path)

if diff == False:

pass

elif diff == True:

accepted = True

return node\_list, start\_x, start\_y, end\_x, end\_y

def winScreen(time\_taken):

while True:

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == KEYDOWN:

if event.key == K\_r:

SolvingInit()

if event.key == K\_ESCAPE:

Main()

screen.fill(BLACK)

message = f'Maze Solved in time: {time\_taken} seconds'

restart = 'R to Return to Menu'

leave = 'ESC to exit to Main Menu' #as of right now, this closes the application however in a alpha build this will return back to the menu

textM = font.render(message, True, font\_color)

textR = font.render(restart, True, font\_color)

textL = font.render(leave, True, font\_color)

screen.blit(textM, (200, 200))

screen.blit(textR, (200, 300))

screen.blit(textL, (200, 400))

pygame.display.update()

def solveMenu():

while True:

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == KEYDOWN:

if event.key == K\_ESCAPE:

Main()

if event.key == K\_1:

return 10

if event.key == K\_2:

return 20

if event.key == K\_3:

return 40

screen.fill(BLACK)

screen.blit(solve\_title, (150,100))

screen.blit(solve\_menu, (150, 300))

pygame.display.update()

def mainSolve():

pack = mazeVerification()

player.x, player.y = 0,0

start\_time = pygame.time.get\_ticks()

if solveMaze(pack, start\_time, passed\_time=0) == True:

time\_taken = timer/1000

winScreen(time\_taken)

def SolvingInit():

size = solveMenu()

global rows, columns, p\_width, p\_height

rows = size

columns = size

p\_width = width/rows

p\_height = height/columns

mainSolve()

#MAZE CREATION CODE

class Player:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def draw(self, colour=GREEN):

pygame.draw.rect(screen, colour, (self.x\*p\_width+2, self.y\*p\_width+2, p\_width-2, p\_width-2))

class grid:

def \_\_init\_\_(self, x, y):

self.x = x #in the form of rows and columns NOT the pygame pixels (have to \*p\_width to get pixel position)

self.y = y

self.wall = True #true if this node is a wall

self.neighbours = []

self.hx = 0 #Heuristic

self.gx = float("inf") #weight of edge

self.fx = float("inf") #Heuristic + Weight of Edge

def draw(self, colour):

#pygame.draw.rect(screen, outline, (self.x\*p\_width, self.y\*p\_width, p\_width, p\_width))

pygame.draw.rect(screen, colour, (self.x\*p\_width+2, self.y\*p\_width+2, p\_width-2, p\_width-2))

def get\_neighbours(self, node\_list):

if self.wall == False:

for node in node\_list: #checking to see whether it has the same coordinates

if node.wall == False:

if self.y > 0: #up node

if self.y - 1 == node.y and self.x == node.x:

self.neighbours.append(node)

if self.x > 0: #left node

if self.y == node.y and self.x - 1 == node.x:

self.neighbours.append(node)

if self.x < rows - 1: #right node

if self.y == node.y and self.x + 1 == node.x:

self.neighbours.append(node)

if self.y < columns - 1: #down node

if self.y + 1 == node.y and self.x == node.x:

self.neighbours.append(node)

def heuristic(self, goal):

g\_x, g\_y = goal.x, goal.y

hsquared = (self.x - g\_x)\*\*2 + (self.y - g\_y)\*\*2

h = (hsquared)\*\*0.5

self.hx = h

#working out the euclidian distance

def createMenu():

while True:

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == KEYDOWN:

if event.key == K\_ESCAPE:

Main()

if event.key == K\_1:

return 10

if event.key == K\_2:

return 15

if event.key == K\_3:

return 20

screen.fill(BLACK)

screen.blit(create\_logo, (150,100))

screen.blit(create\_menu, (150,300))

pygame.display.update()

def nodeInit():

node\_list = []

walls = []

for i in range(rows):

for j in range(columns):

block = grid(i, j)

node\_list.append(block)

walls.append(block)

return node\_list, walls

def createMaze(node\_list, walls):

count = 0

while True:

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == MOUSEBUTTONDOWN:

pos = pygame.mouse.get\_pos()

mx, my = pos

mx, my = int(mx//p\_width), int(my//p\_width)

if mx < rows and my < columns:

for i in node\_list:

if i.x == mx and i.y == my:

node = i

if event.button == 1: #click to remove wall, click again to add wall

if node.wall == True:

node.wall = False

walls.remove(node)

count += 1

elif node.wall == False:

node.wall = True

walls.append(node)

count -= 1

#print(node)

if event.type == KEYDOWN:

if event.key == K\_ESCAPE:

CreationInit()

if event.key == K\_c:

if count >= 2:

return True

screen.fill(BLACK)

text1 = font\_hints.render('Use Mouse Clicks to', True, font\_white)

text2 = font\_hints.render('interact with the walls', True, font\_white)

text3 = font\_hints.render('Press ESC key to', True, font\_white)

text4 = font\_hints.render('Return to Menu', True, font\_white)

text5 = font\_hints.render('Press C key to', True, font\_white)

text6 = font\_hints.render('Confirm/Continue', True, font\_white)

screen.blit(text1, (width+20, 100))

screen.blit(text2, (width+20, 130))

screen.blit(text3, (width+20, 200))

screen.blit(text4, (width+20, 230))

screen.blit(text5, (width+20, 300))

screen.blit(text6, (width+20, 330))

for i in walls:

i.draw(WHITE)

pygame.display.update()

def addStartEnd(node\_list, walls, List):

while True:

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == MOUSEBUTTONDOWN:

pos = pygame.mouse.get\_pos()

mx, my = pos

mx, my = int(mx//p\_width), int(my//p\_width)

if mx < rows and my < columns:

for i in node\_list:

if i.x == mx and i.y == my:

node = i

if node.wall == False and node not in Start: #if the block chosen is not a wall and it is not the starting node

if len(List) != 0:

List.remove(List[0])

List.append(node)

node.wall = False

if event.type == KEYDOWN:

if event.key == K\_ESCAPE:

CreationInit()

if event.key == K\_c and len(List) == 1:

return True

#return True

screen.fill(BLACK)

text1 = font\_hints.render('Use Mouse Clicks to', True, font\_white)

text2 = font\_hints.render('add a Start/End Square', True, font\_white)

text3 = font\_hints.render('Blue is the Start', True, font\_blue)

text4 = font\_hints.render('Red is the Goal', True, font\_red)

text5 = font\_hints.render('Press ESC key to', True, font\_white)

text6 = font\_hints.render('Return to Menu', True, font\_white)

text7 = font\_hints.render('Press C key to', True, font\_white)

text8 = font\_hints.render('Confirm/Continue', True, font\_white)

screen.blit(text1, (width+20, 100))

screen.blit(text2, (width+20, 130))

screen.blit(text3, (width+20, 160))

screen.blit(text4, (width+20, 190))

screen.blit(text5, (width+20, 260))

screen.blit(text6, (width+20, 290))

screen.blit(text7, (width+20, 360))

screen.blit(text8, (width+20, 390))

for i in walls:

i.draw(WHITE)

if len(Start) > 0:

Start[0].draw(BLUE)

if len(End) > 0:

End[0].draw(RED)

pygame.display.update()

def aStar(node\_list, start\_node, goal\_node):

graph = {}

for node in node\_list:

node.heuristic(goal\_node)

node.get\_neighbours(node\_list)

if node.wall == False:

graph[node] = node.neighbours

shortest\_path = []

path = {}

start\_node.fx = 0

start\_node.gx = 0

while len(graph) > 0:

smallestfx = [node for node in graph]

smallestfx.sort(key=lambda x: x.fx, reverse=False)

smallest = smallestfx[0] #the node with the smallest fx value

for neighbour in graph[smallest]: #for the neighbours of the smallest node

if neighbour in graph and neighbour.fx > 1 + smallest.gx + neighbour.hx:

neighbour.gx = 1 + smallest.gx

neighbour.fx = neighbour.gx + neighbour.hx

path[neighbour] = smallest

graph.pop(smallest)

node = goal\_node

while node != start\_node:

shortest\_path.insert(0, node)

node = path[node]

shortest\_path.insert(0, start\_node)

return shortest\_path

def createCheckWall(node\_list, node, direction):

for i in node\_list:

if direction == 'u':

if i.x == node.x and i.y == node.y-1:

if i.wall == False:

return False #no wall in the intended direction

else:

return True

elif direction == 'd':

if i.x == node.x and i.y == node.y+1:

if i.wall == False:

return False #no wall in the intended direction

else:

return True

elif direction == 'l':

if i.x == node.x-1 and i.y == node.y:

if i.wall == False:

return False #no wall in the intended direction

else:

return True

elif direction == 'r':

if i.x == node.x+1 and i.y == node.y:

if i.wall == False:

return False #no wall in the intended direction

else:

return True

def solveComputer(walls, node\_list, start\_node, goal\_node):

try:

path = aStar(node\_list, start\_node, goal\_node)

while True:

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == KEYDOWN:

if event.key == K\_ESCAPE:

CreationInit()

if event.key == K\_RETURN:

return True

screen.fill(BLACK)

text1 = font\_hints.render('Press RETURN Key to', True, font\_white)

text2 = font\_hints.render('finish viewing the maze', True, font\_white)

text3 = font\_hints.render('Press ESC key to', True, font\_white)

text4 = font\_hints.render('Return to Menu', True, font\_white)

screen.blit(text1, (width+15, 200))

screen.blit(text2, (width+15, 230))

screen.blit(text3, (width+15, 300))

screen.blit(text4, (width+15, 330))

for i in walls:

i.draw(WHITE)

for j in path:

j.draw(YELLOW)

pygame.display.update()

except KeyError:

print("no path exists -- error, restarting application...")

CreationInit()

def solveUser(walls, node\_list, start\_node, goal\_node, start\_time):

player = Player(start\_node.x, start\_node.y)

while True:

global timer

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == KEYDOWN:

if event.key == K\_ESCAPE:

CreationInit()

if event.key == K\_UP:

if createCheckWall(node\_list, player, 'u') == False:

player.y -= 1

if event.key == K\_LEFT:

if createCheckWall(node\_list, player, 'l') == False:

player.x -= 1

if event.key == K\_RIGHT:

if createCheckWall(node\_list, player, 'r') == False:

player.x += 1

if event.key == K\_DOWN:

if createCheckWall(node\_list, player, 'd') == False:

player.y += 1

screen.fill(BLACK)

text1 = font\_hints.render('Use Directional Keys to', True, font\_white)

text2 = font\_hints.render('navigate the maze', True, font\_white)

text3 = font\_hints.render('Press ESC key to', True, font\_white)

text4 = font\_hints.render('Return to Menu', True, font\_white)

screen.blit(text1, (width+20, 200))

screen.blit(text2, (width+20, 230))

screen.blit(text3, (width+20, 300))

screen.blit(text4, (width+20, 330))

for i in walls:

i.draw(WHITE)

Start[0].draw(BLUE)

End[0].draw(RED)

player.draw()

if int(player.x) == int(goal\_node.x) and int(player.y) == int(goal\_node.y):

return True

else:

passed\_time = pygame.time.get\_ticks() - start\_time #in milliseconds

timer = passed\_time

time = str(passed\_time/1000)

text = font.render(f'Time: {time}', True, font\_color)

screen.blit(text,(width+20, 100))

pygame.display.update()

def UserOrComputer():

while True:

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == KEYDOWN:

if event.key == K\_ESCAPE:

CreationInit()

if event.key == K\_1:

return True #Computer

if event.key == K\_2:

return False #User

screen.fill(BLACK)

screen.blit(create\_dTitle, (150, 100))

screen.blit(create\_decision, (150, 300))

pygame.display.update()

def winScreenUser(time\_taken):

while True:

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == KEYDOWN:

if event.key == K\_r:

CreationInit()

if event.key == K\_ESCAPE:

Main()

screen.fill(BLACK)

message = f'Maze Solved in time: {time\_taken} seconds'

restart = 'R to Return to Menu'

leave = 'ESC to exit to Main Menu'

textM = font.render(message, True, font\_color)

textR = font.render(restart, True, font\_color)

textL = font.render(leave, True, font\_color)

screen.blit(textM, (200, 200))

screen.blit(textR, (200, 300))

screen.blit(textL, (200, 400))

pygame.display.update()

def winScreenComputer():

while True:

clock.tick(60)

for event in pygame.event.get():

if event.type == pygame.QUIT:

sys.exit()

if event.type == KEYDOWN:

if event.key == K\_r:

CreationInit()

if event.key == K\_ESCAPE:

Main()

screen.fill(BLACK)

restart = 'R to Return to Menu'

leave = 'ESC to exit to Main Menu'

textR = font.render(restart, True, font\_color)

textL = font.render(leave, True, font\_color)

screen.blit(textR, (300, 300))

screen.blit(textL, (300, 400))

pygame.display.update()

def Computer(walls, node\_list, start\_node, goal\_node):

if solveComputer(walls, node\_list, start\_node, goal\_node) == True:

winScreenComputer()

def User(walls, node\_list, start\_node, goal\_node):

start\_time = pygame.time.get\_ticks()

if solveUser(walls, node\_list, start\_node, goal\_node, start\_time) == True:

time\_taken = timer/1000

winScreenUser(time\_taken)

def mainCreate():

pack = nodeInit()

node\_list = pack[0]

walls = pack[1]

if createMaze(node\_list, walls) == True:

if addStartEnd(node\_list, walls, Start) == True:

if addStartEnd(node\_list, walls, End) == True:

start\_node = Start[0] #this is the global variable that contains the info for the start node

goal\_node = End[0] #same as above

if UserOrComputer() == True:

Computer(walls, node\_list, start\_node, goal\_node)

else:

User(walls, node\_list, start\_node, goal\_node)

def CreationInit():

size = createMenu()

global rows, columns, p\_width, walls, Start, End

walls = []

Start = []

End = []

rows = size

columns = size

p\_width = width//rows

mainCreate()

if \_\_name\_\_ == '\_\_main\_\_':

Main()