MAZE GENERATOR & SOLVER

Submitted in partial fulfilment of the requirements

of the degree of

Bachelor of Engineering in

Artificial Intelligence and Data Science

by

Harshita Anala –2

Shruti Devlekar – 9

Akshiti Kachhawah –22

Abhishek Thorat – 61

Department of Artificial Intelligence and Data Science

Vivekanand Education Society's Institute of Technology

2021-2022

Vivekanand Education Society's

Institute of Technology

(Affiliated to University of Mumbai, Approved by AICTE & Recognized by Govt. of Maharashtra)

Since 1962

Department of Artificial Intelligence and Data Science

CERTIFICATE

This is to certify that **Mr. Abhishek Thorat, Ms.Shruti Devlekar,Ms. Harshita Anala, Ms.Akshiti Kachhawah** of Second Year of Artificial Intelligence and Data Science studying under the University of Mumbai have satisfactorily presented the Mini Project entitled **MAZE GENERATOR & SOLVER** as a part of the MINI-PROJECT for Semester-III under the guidance of ***Mrs. Sangeeta Oswal*** *in the year* ***2021-2022.***

Date: 4 May 2022

Oswal

(Name and sign)
Head of Department
Supervisor/Guide

(Name and sign)

Department of Artificial Intelligence and Data Science

DECLARATION

We, ***Mr. Abhishek Thorat, Ms. Shruti Devlekar, Ms. Harshita Anala, Ms. Akshiti Kachhawah*** from ***D6AD***, declares that this project represents our ideas in our own words without plagiarism and wherever others' ideas or words have been included, we have adequately cited and referenced the original sources.

We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our project work.

We declare that we have maintained a minimum 75% attendance, as per the University of Mumbai norms.

We understand that any violation of the above will be cause for disciplinary action by the Institute.

Yours Faithfully

1.Mr. Abhishek Thorat

2.Ms .Shruti Devlekar

3. *Ms. Harshita Anala*

4. Ms. Akshiti Kachhawah

(Name & Signature of Students with Date)

Acknowledgement

We are sincerely grateful to our college, VESIT, our respected principal Dr. J. M. Nair ma'am and our HoD Dr. M. Vijayalaksmi ma'am for giving us the opportunity to work on our Mini project i.e. an AI based game development. Completing this project would not have been possible without their guidance and support. We extend our gratitude to our mentor Mrs. Sangeeta Oswal ma'am who guided us in all the meetings with her. She provided us with some valuable information and resources to help increase the efficiency of our project. We would also like to thank our peers who helped us whenever we faced any difficulty while developing the game and our parents who always supported us in their own ways. Finally, we thank all the mentioned and not mentioned who were a part of our project.

Table of Contents

1. Abstract:

The following report consists of a detailed description of our Mini Project. We have created a classic game - **'Maze Generator and Solver'** in which we have implemented various packages and modules of python and AI as well as basic algorithms.

1.1 Introduction:

We imported the turtle package in python which would help us in implementing graphics in our code. Following the turtle, we imported the choice module from random and the vector and floor modules from the freegames library.

The start of the code represents the initial state of the game/game objects. Then we have multiple user-defined functions .

1. square( ) - which fills the tile with the x and y coordinates.
2. offset( ) - which returns the next index in which the pointer moves
3. valid( ) - which returns True if the next tile is valid for movement.
4. world() - used for Drawing the world, the tiles, and the score points.
5. move( ) - used For controlling the movement of all the pointer
6. change() - change the direction vector of the pointer if it's valid.

Then we use the setup( ) function from turtle to set up the GUI window for our game.

The write( ) function is used to display the score in which state['score'] is used to input the current score of the user.

The game ends when the pointer reaches the end point and hence, the game is over.

1.2 Problem Statement:

Our project title is **Maze Generator** and **Maze Solver**, firstly a random maze will be generated using a randomised algorithm and later we will be solving it as a user and corresponding score will be displayed.

There is a 20*20 matrix and a 2 points start and end . The pointer goes from start to the end with the help of user.The user can move the pointer is the desired direction to reach the end point, avaoiding the walls. maze generator uses prims algorithm and random module to generate new mazes everytime we start the game.

1.3 Objective:

To randomly generate a maze using an AI algorithm which can be solved by the user.

1.4 Scope:

Through this project users will be able to apply skills to solve the maze and and successfully find the path. The concept will be helpful for kids to develop their analytical and logical skills.

2. Literature Survey:

A maze is a path, or a collection of paths, typically from an entrance to a goal. In other words, there is an entrance and an exit and starting from the entrance, you need to navigate through the complex paths and walls and find a path towards the exit.
Randomized Prim's Algorithm consists of the following steps:

Start with a grid full of walls

Pick a cell, mark it as part of the maze. Add the walls of the cell to the walls of the list

While there are walls in the list:

1. Pick a random wall from the list. If only one of the two cells that the wall divides is visited, then:

a) Make the wall a passage and mark the unvisited cell as part of the maze

b) Add the neighboring walls of the cell to the wall list.

2. Remove the wall from the list

Everytime a new maze is generated on the basis of an algorithm.

Prim's Algorithm:

Prim's algorithm (also known as Jarník's algorithm) is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

How Prim's algorithm works:

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.

2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

Randomized Prim's Algorithm consists of the following steps:

Start with a grid full of walls

Pick a cell, mark it as part of the maze. Add the walls of the cell to the walls of the list

While there are walls in the list:

1. Pick a random wall from the list. If only one of the two cells that the wall divides is visited, then:

a) Make the wall a passage and mark the unvisited cell as part of the maze

b) Add the neighboring walls of the cell to the wall list.

2. Remove the wall from the list.

2.2 Papers/Findings :

Following articles and paper were used:

1. https://medium.com/swlh/fun-with-python-1-maze-generator-931639b4fb7e
2. https://en.wikipedia.org/wiki/Prim%27s_algorithm
3. https://en.wikipedia.org/wiki/Maze_generation_algorithm
4. 3.5 Prims and Kruskals Algorithms - Greedy Method

3. Analysis and Design:

3.1 Implementation Details:

1. The program consists of a 20x20 maze which is available to the user.
2. There will be two ends: start and end.
3. A pointer will start from the Start and has to reach the end point guided by the user.
4. In this process the score will be calculated simultaneously.
5. After the pointer reaches the end point a message game over is displayed..
6. The maze is derived using Prim's algorithm.

3.2 Proposed Solution:

The maze solution will be derived using the Prim's algorithm.

Start with a grid full of walls

Pick a cell, mark it as part of the maze. Add the walls of the cell to the walls of the list

While there are walls in the list:

1. Pick a random wall from the list. If only one of the two cells that the wall divides is visited, then:

a) Make the wall a passage and mark the unvisited cell as part of the maze

b) Add the neighboring walls of the cell to the wall list.

2. Remove the wall from the list

4. Results and Discussion

4.1 Implementation details

We created two python files

For Backend(Maze Generator): idk.py

```
import random

import time

from colorama import init

from colorama import Fore, Back, Style




# Functions

def printMaze(maze):

    for i in range(0, height):

        for j in range(0, width):

            if (maze[i][j] == 'u'):

                #print(Fore.WHITE + str(maze[i][j]), end=" ")

                pass

            elif (maze[i][j] == 1):

                #print(Fore.GREEN + str(maze[i][j]), end=" ")

                pass
            else:
                #print(Fore.RED + str(maze[i][j]), end=" ")
                pass
        print('\n')



# Find number of surrounding cells
def surroundingCells(rand_wall):
    s_cells = 0
    if (maze[rand_wall[0] - 1][rand_wall[1]] == 1):
        s_cells += 1
    if (maze[rand_wall[0] + 1][rand_wall[1]] == 1):
        s_cells += 1
    if (maze[rand_wall[0]][rand_wall[1] - 1] == 1):
        s_cells += 1
    if (maze[rand_wall[0]][rand_wall[1] + 1] == 1):
        s_cells += 1

    return s_cells



# Main code
# Init variables
wall = 0
cell = 1
unvisited = 'u'
```

```python
height = 20

width = 20

maze = []


# Initialize colorama

init()


# Denote all cells as unvisited

for i in range(0, height):

   line = []

   for j in range(0, width):

       line.append(unvisited)

   maze.append(line)


# Randomize starting point and set it a cell

starting_height = int(random.random() * height)

starting_width = int(random.random() * width)

if (starting_height == 0):

   starting_height += 1

if (starting_height == height - 1):

   starting_height -= 1

if (starting_width == 0):

   starting_width += 1

if (starting_width == width - 1):

   starting_width -= 1
```

```
# Mark it as cell and add surrounding walls to the list
maze[starting_height][starting_width] = cell
walls = []
walls.append([starting_height - 1, starting_width])
walls.append([starting_height, starting_width - 1])
walls.append([starting_height, starting_width + 1])
walls.append([starting_height + 1, starting_width])

# Denote walls in maze
maze[starting_height - 1][starting_width] = 0
maze[starting_height][starting_width - 1] = 0
maze[starting_height][starting_width + 1] = 0
maze[starting_height + 1][starting_width] = 0

while (walls):
   # Pick a random wall
   rand_wall = walls[int(random.random() * len(walls)) - 1]

   # Check if it is a left wall
   if (rand_wall[1] != 0):
       if (maze[rand_wall[0]][rand_wall[1] - 1] == 'u' and
maze[rand_wall[0]][rand_wall[1] + 1] == 1):
           # Find the number of surrounding cells
           s_cells = surroundingCells(rand_wall)

           if (s_cells < 2):
```

```
                    # Denote the new path
                    maze[rand_wall[0]][rand_wall[1]] = 1


                    # Mark the new walls
                    # Upper cell
                    if (rand_wall[0] != 0):
                        if (maze[rand_wall[0] - 1][rand_wall[1]] !=
1):
                            maze[rand_wall[0] - 1][rand_wall[1]] =
0
                        if ([rand_wall[0] - 1, rand_wall[1]] not in
walls):
                            walls.append([rand_wall[0] - 1,
rand_wall[1]])


                    # Bottom cell
                    if (rand_wall[0] != height - 1):
                        if (maze[rand_wall[0] + 1][rand_wall[1]] !=
1):
                            maze[rand_wall[0] + 1][rand_wall[1]] =
0
                        if ([rand_wall[0] + 1, rand_wall[1]] not in
walls):
                            walls.append([rand_wall[0] + 1,
rand_wall[1]])


                    # Leftmost cell
                    if (rand_wall[1] != 0):
```

```
                    if (maze[rand_wall[0]][rand_wall[1] - 1] !=
1):
                        maze[rand_wall[0]][rand_wall[1] - 1] =
0
                    if ([rand_wall[0], rand_wall[1] - 1] not in
walls):
                        walls.append([rand_wall[0],
rand_wall[1] - 1])

            # Delete wall
            for wall in walls:
                if (wall[0] == rand_wall[0] and wall[1] ==
rand_wall[1]):
                    walls.remove(wall)

            continue

    # Check if it is an upper wall
    if (rand_wall[0] != 0):
        if (maze[rand_wall[0] - 1][rand_wall[1]] == 'u' and
maze[rand_wall[0] + 1][rand_wall[1]] == 1):

            s_cells = surroundingCells(rand_wall)
            if (s_cells < 2):
                # Denote the new path
                maze[rand_wall[0]][rand_wall[1]] = 1

                # Mark the new walls
                # Upper cell
                if (rand_wall[0] != 0):
                    if (maze[rand_wall[0] - 1][rand_wall[1]] !=
1):
                        maze[rand_wall[0] - 1][rand_wall[1]] =
0
                    if ([rand_wall[0] - 1, rand_wall[1]] not in
walls):
                        walls.append([rand_wall[0] - 1,
rand_wall[1]])


                # Leftmost cell
                if (rand_wall[1] != 0):
                    if (maze[rand_wall[0]][rand_wall[1] - 1] !=
1):
                        maze[rand_wall[0]][rand_wall[1] - 1] =
0
                    if ([rand_wall[0], rand_wall[1] - 1] not in
walls):
                        walls.append([rand_wall[0],
rand_wall[1] - 1])


                # Rightmost cell
                if (rand_wall[1] != width - 1):
                    if (maze[rand_wall[0]][rand_wall[1] + 1] !=
1):
                        maze[rand_wall[0]][rand_wall[1] + 1] =
0
                    if ([rand_wall[0], rand_wall[1] + 1] not in
walls):
```

```
                        walls.append([rand_wall[0],
rand_wall[1] + 1])


            # Delete wall

            for wall in walls:

                if (wall[0] == rand_wall[0] and wall[1] ==
rand_wall[1]):

                    walls.remove(wall)


            continue


    # Check the bottom wall

    if (rand_wall[0] != height - 1):

        if (maze[rand_wall[0] + 1][rand_wall[1]] == 'u' and
maze[rand_wall[0] - 1][rand_wall[1]] == 1):


            s_cells = surroundingCells(rand_wall)

            if (s_cells < 2):

                # Denote the new path

                maze[rand_wall[0]][rand_wall[1]] = 1


                # Mark the new walls

                if (rand_wall[0] != height - 1):

                    if (maze[rand_wall[0] + 1][rand_wall[1]] !=
1):

                        maze[rand_wall[0] + 1][rand_wall[1]] =
0
```

```
                    if ([rand_wall[0] + 1, rand_wall[1]] not in
walls):
                        walls.append([rand_wall[0] + 1,
rand_wall[1]])
                if (rand_wall[1] != 0):
                    if (maze[rand_wall[0]][rand_wall[1] - 1] !=
1):
                        maze[rand_wall[0]][rand_wall[1] - 1] =
0
                    if ([rand_wall[0], rand_wall[1] - 1] not in
walls):
                        walls.append([rand_wall[0],
rand_wall[1] - 1])
                if (rand_wall[1] != width - 1):
                    if (maze[rand_wall[0]][rand_wall[1] + 1] !=
1):
                        maze[rand_wall[0]][rand_wall[1] + 1] =
0
                    if ([rand_wall[0], rand_wall[1] + 1] not in
walls):
                        walls.append([rand_wall[0],
rand_wall[1] + 1])

            # Delete wall
            for wall in walls:
                if (wall[0] == rand_wall[0] and wall[1] ==
rand_wall[1]):
                    walls.remove(wall)

            continue
```

```
   # Check the right wall
   if (rand_wall[1] != width - 1):
       if (maze[rand_wall[0]][rand_wall[1] + 1] == 'u' and
maze[rand_wall[0]][rand_wall[1] - 1] == 1):

           s_cells = surroundingCells(rand_wall)
           if (s_cells < 2):
               # Denote the new path
               maze[rand_wall[0]][rand_wall[1]] = 1

               # Mark the new walls
               if (rand_wall[1] != width - 1):
                   if (maze[rand_wall[0]][rand_wall[1] + 1] !=
1):
                       maze[rand_wall[0]][rand_wall[1] + 1] =
0
                   if ([rand_wall[0], rand_wall[1] + 1] not in
walls):
                       walls.append([rand_wall[0],
rand_wall[1] + 1])
               if (rand_wall[0] != height - 1):
                   if (maze[rand_wall[0] + 1][rand_wall[1]] !=
1):
                       maze[rand_wall[0] + 1][rand_wall[1]] =
0
                   if ([rand_wall[0] + 1, rand_wall[1]] not in
walls):
```

```
                        walls.append([rand_wall[0] + 1,
rand_wall[1]])
                if (rand_wall[0] != 0):
                    if (maze[rand_wall[0] - 1][rand_wall[1]] !=
1):
                        maze[rand_wall[0] - 1][rand_wall[1]] =
0
                    if ([rand_wall[0] - 1, rand_wall[1]] not in
walls):
                        walls.append([rand_wall[0] - 1,
rand_wall[1]])


            # Delete wall
            for wall in walls:
                if (wall[0] == rand_wall[0] and wall[1] ==
rand_wall[1]):
                    walls.remove(wall)


            continue


    # Delete the wall from the list anyway
    for wall in walls:
        if (wall[0] == rand_wall[0] and wall[1] ==
rand_wall[1]):
            walls.remove(wall)


# Mark the remaining unvisited cells as walls
for i in range(0, height):
    for j in range(0, width):

        if (maze[i][j] == 'u'):

            maze[i][j] = 0


# Set entrance and exit

for i in range(0, width):

    if (maze[1][i] == 1):

        maze[0][i] = 1

        break


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

    if (maze[height - 2][i] == 1):

        maze[height - 1][i] = 1

        break


# Print final maze

printMaze(maze)

print(maze)
```

:

Code:

For GUI :

```
# Importation

from random import choice

from turtle import *
```

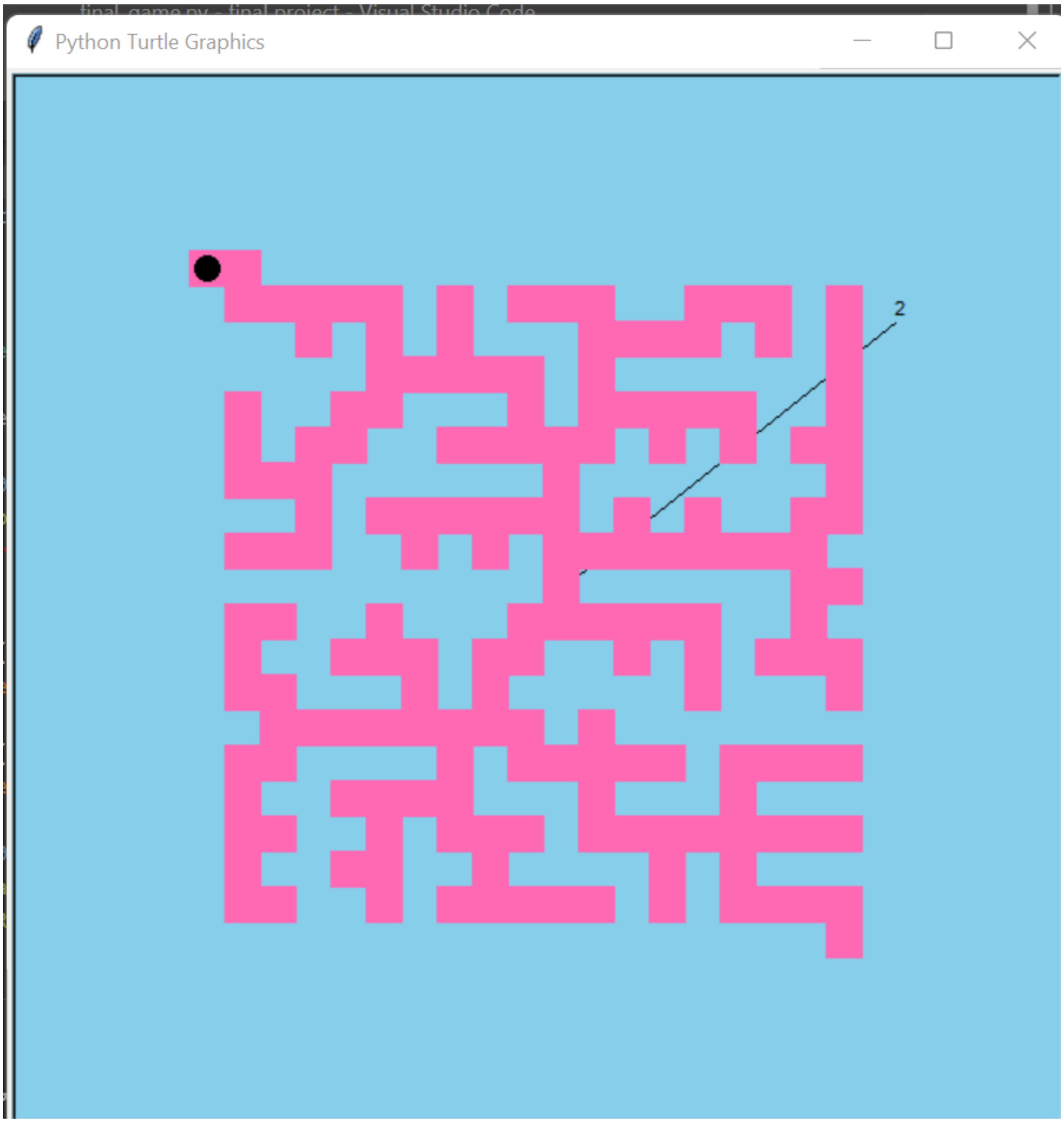
```
import numpy as np

from freegames import floor, vector


import idk  # Random Maze Code


# Initialisation


maze = idk.maze

hehe = np.array(maze)

hehe = hehe.flatten()

# print(maze)




# print(hehe)

state = {'score': 0}


path = Turtle(visible=False)

writer = Turtle(visible=False)


aim = vector(5, 0)  # Velocity = {in Pos X} 5px per sec

our_user = vector(-180, 180)  # Spawn Location

opponents = [  # Opponents and their Velocity and Spawn
Locations(if req)

    # [vector(-180, 160), vector(5, 0)],

    # [vector(-180, -160), vector(0, 5)],
    # [vector(100, 160), vector(0, -5)],

    # [vector(100, -160), vector(-5, 0)],

]


# updating the spawn location of the user to make sure it is
not out of bonds.

hehe[0] = hehe[1] = 1

tiles = hehe  # tiles is the map of our game in 1d (20x20)



'''

Sample Map

tiles = [

    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

    0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

    0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0,

    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

    0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0,

    0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0,

    0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,

    0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0,

    0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

    0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0,

    0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0,

    0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,

    0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
    0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0,
    0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0,
    0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0,
    0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
]
'''



def world():
    bgcolor('cyan')
    path.color('pink')

    for index in range(len(tiles)):
        tile = tiles[index]

        if tile > 0:
            x = (index % 20) * 20 - 200
            y = 180 - (index // 20) * 20
            square(x, y)   # Draw Square Path



def square(x, y):
    path.up()
    path.goto(x, y)
    path.down()
    path.begin_fill()

    for count in range(4):
        path.forward(20)
        path.left(90)

    path.end_fill()


def offset(point):
    x = (floor(point.x, 20) + 200) / 20
    y = (180 - floor(point.y, 20)) / 20
    index = int(x + y * 20)
    return index


def valid(point):
    index = offset(point)

    try:
        if tiles[index] == 0:
            return False
    except IndexError:
        # Game Over Screen
        writer.goto(-150, 100)

        writer.write('Your Score was : ' +

                     str(state['score']), font=("Times New
Roman", 30, "italic"))


        writer.goto(-200, -50)

        writer.write('Game Over!', font=(

            "Times New Roman", 60, "bold"))


        done()


    index = offset(point + 19)


    if tiles[index] == 0:

        return False


    return point.x % 20 == 0 or point.y % 20 == 0



def move():

    writer.undo()

    writer.write(state['score'])


    clear()
```

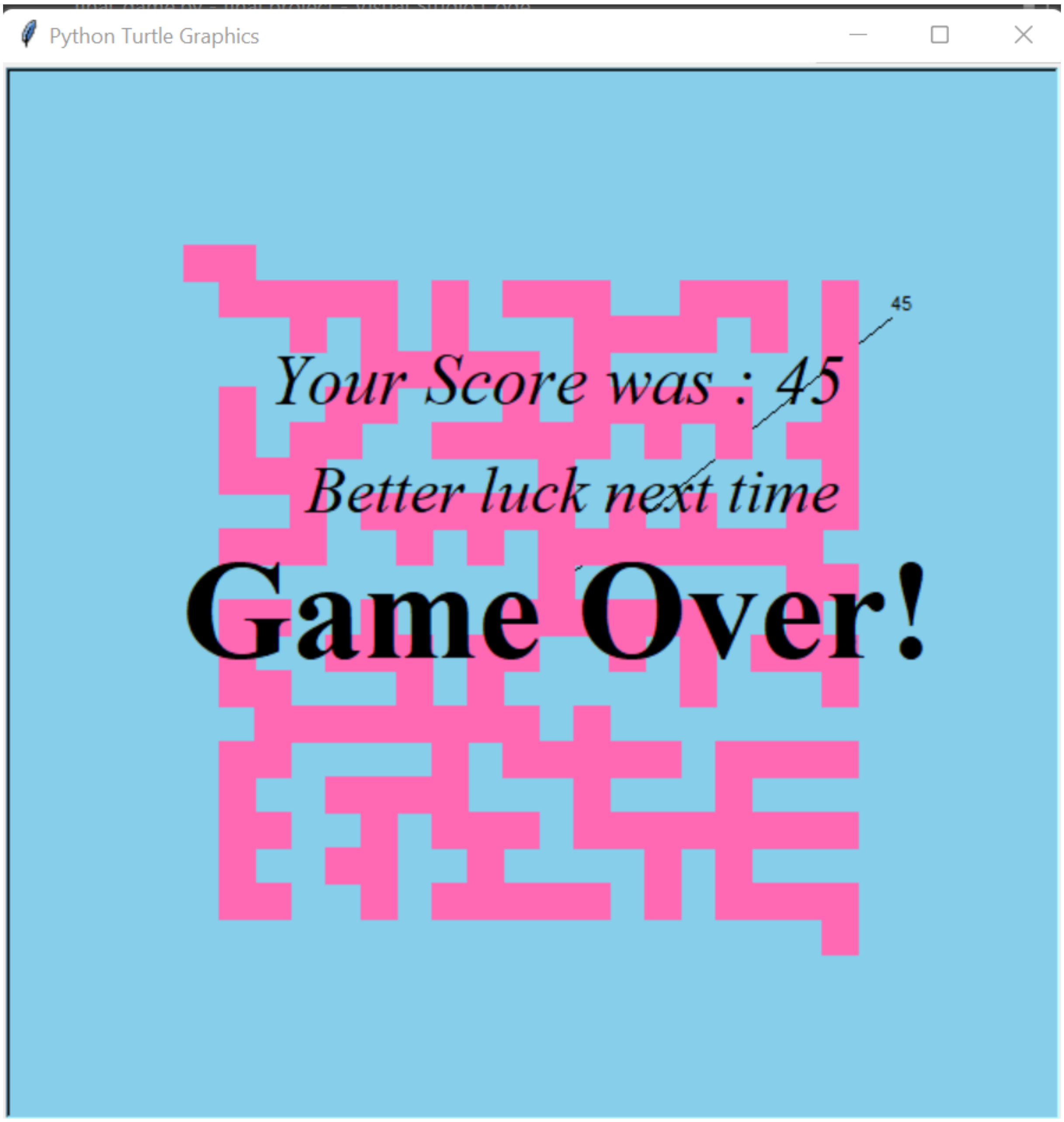
```
if valid(our_user + aim):
    our_user.move(aim)

index = offset(our_user)

if tiles[index] == 1:
    tiles[index] = 2
    state['score'] += 1
    x = (index % 20) * 20 - 200
    y = 180 - (index // 20) * 20
    square(x, y)

up()
goto(our_user.x + 10, our_user.y + 10)
dot(15, 'black')

for point, course in opponents:
    if valid(point + course):
        point.move(course)
    else:
        options = [
            vector(5, 0),
            vector(-5, 0),
            vector(0, 5),
            vector(0, -5),
        ]
```

```
            plan = choice(options)
            course.x = plan.x
            course.y = plan.y

        up()
        goto(point.x + 10, point.y + 10)
        dot(20, 'red')

    update()

    # for point, course in opponents:
    #     if abs(our_user - point) < 20:
    #         return

    ontimer(move, 100)


def change(x, y):
    if valid(our_user + vector(x, y)):
        aim.x = x
        aim.y = y


setup(600, 600, 500, 120)
```

```
tracer(False)
writer.goto(200, 160)
writer.color('black')

hideturtle()
writer.write(state['score'])

listen()
onkey(lambda: change(5, 0), 'Right')
onkey(lambda: change(-5, 0), 'Left')
onkey(lambda: change(0, 5), 'Up')
onkey(lambda: change(0, -5), 'Down')
world()
move()

done()
```

4.2 Working Model:





Your Score was : 45

Better luck next time

Game Over!

5. Conclusion and Future Work:

Conclusion:

Different types of maze-solving algorithms are often needed

Here we used Prim's algorithm to do the same.

In summary, we built an maze game based on the children's game Labyrinth

Our original design goals were to avoid complexity, to make our design scalable and to make our design easily mutable. Since we were able to add extensions easily to our project, we believe that we were fairly successful in meeting these objectives.

Future Work:

At present, video games appear to be resources that are more and more promising regarding their use in primary education thanks to their high interactivity and visual attractiveness, in addition to the ability to incorporate knowledge, didactic tasks and virtual objects in an appealing and fascinating way. Educational video mazes represent a specific type of puzzle game, where the environment has (unlike simple mazes) a complex branching with options of choosing the traversal path and direction, and may have multiple entrances, exits and dead ends. The research provides some of the results from a study of the potential of educational maze games for teaching in primary schools. The aim is to clarify the teachers' needs and preferences for such games specially designed for primary education. The educational video mazes have to be described formally and next generated by an open software platform in an automatic and straightforward way.