TRƯỜNG ĐẠI HỌC
SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH
HCMC University of Technology and Education
HCMUTE

KHOA ĐÀO TẠO QUỐC TẾ

FINAL REPORT

***

DESIGN AND IMPLEMENT A MAZE GAME USING AI ALGORITHMS

MÃ MÔN HỌC: DASA230179E_02FIE

GRADE: Monday, period 3-5

STUDENT PERFORMENT:

Full name	Student ID	Contribution
Đỗ Huỳnh Bảo Đăng	21110764	100%
Nguyễn Duy Mạnh	21110780	100%
Đỗ Xuân Trường	21110101	100%

LECTURER: PGS.TS. Hoàng Văn Dũng

Ho Chi Minh City, November 28, 2023

LIST OF MEMBERS INVOLVED IN WRITING REPORTS

NUM	FULL NAME	STUDENT ID	COMPLETION
1	Đỗ Huỳnh Bảo Đăng	21110764	100%
2	Nguyễn Duy Mạnh	21110780	100%
3	Đỗ Xuân Trường	21110101	100%

Note:

- Percentage = 100%: The percentage level of each student participating.
- Leader: Đỗ Huỳnh Bảo Đăng

Lecturer reviews

…….

…….

…….

…….

28/11/2023

Score

(Signature of lecturer)

Hoàng Văn Dũng

LIST OF ABBREVIATIONS

BFS	Breadth first search
DFS	Depth-First Search
GBFS	Greedy best first search

THANKS

For successfully completing this project and report, we would like to express our sincere thanks to the lecturer, Associate Professor Dr. Hoang Van Dung, who directly supported us during the working process. We thank you for giving us advice from our practical experience to guide us in accordance with the requirements of the chosen topic, always answering questions and giving suggestions and corrections in time to help us overcome our weaknesses and complete well as on time.

We would also like to sincerely thank the teachers in the Faculty of International Education in general and the Information Technology industry in particular for their dedication to imparting the necessary knowledge to help us have the foundation to make this topic, for creating conditions for us to learn and implement the topic well. Along with that, we would like to thank our classmates for providing a lot of useful information and knowledge to help us improve our topic.

The topic and report are carried out in a short period of time, with limited knowledge and many other limitations in terms of technique and experience in implementing a software project. Therefore, in the process of creating the topic, there are inevitable shortcomings, so we look forward to receiving valuable comments from teachers so that our knowledge is improved and we can do better in the next time. We sincerely thank you

Finally, we wish you good health and success in your career. Once again we would like to sincerely thank you.

Ho Chi Minh City, November 28, 2023

Đỗ Huỳnh Bảo Đăng
Nguyễn Duy Mạnh
Đỗ Xuân Trường

INDEX

LIST OF FIGURES

1. INTRODUCTION

1.1 Reason for choice:

- Maze game are relatively simple game, making them suitable for a focused AI project. The confined nature of mazes allows students to concentrate on fundamental AI concepts without being overwhelmed by unnecessary complexity
- Maze solving is a classic problem that involves search algorithms. Implementing algorithms like depth-first search, breadth-first search, A* search, or other heuristic-based approaches allows us to understand and experiment with fundamental AI search techniques.

1.2 Statement of the problem:

The specific problem is to use algorithms learned in artificial intelligence to decode a maze and find the way to the treasure location.

1.3 Purpose and requirements to be fulfilled:

The purpose of this topic is to use a visual interface to describe the process of decoding the algorithms AStar, Breadth first search, Depth first search, Dijkstra'algorithm, Greedy best first search algorithm and calculate the execution time of each algorithm. for each different state space.

1.4 Scope and objects:

The problem of finding a path in a maze is used to simulate and evaluate algorithms in decoding different mazes, thereby making comments and conclusions about decoding the maze and evaluating the state space. as well as calculate the algorithm execution time to determine for each different state space, which algorithm we need to use to optimize as much as possible.

This large exercise is valuable as a simulation for future educational purposes.

2. THEORETICAL BASIS

2.1 Programming environment

For this big assignment, we use python as the main language to write code with the visual studio code compiler and use the pygame library to create the user interface.

2.2 Support library

We use simple libraries in python to do things like random, time, sys, os, ... use csv files to create mazes and mostly apply pygame to create interfaces and create events of buttons.

2.3 Basic theory of AI algorithms

2.3.1 Breadth first search (BFS):

The Breadth-First Search (BFS) algorithm is a method for traversing a graph breadthwise from a starting vertex. It uses a queue to keep track of the vertices that need to be traversed and ensures that vertices near the starting vertex are traversed first. BFS is suitable for finding the shortest path and is widely used in applications such as network optimization, finding opponents in games, and graph traversal. The time complexity of BFS is $O(V+E)$, với V là số đỉnh và E là số cạnh của đồ thị.

2.3.2 Depth first search (DFS):

The Depth-First Search (DFS) algorithm is a method for traversing a graph in depth from a starting vertex. It uses a stack to keep track of vertices to visit, diving deep into one branch of the graph before going back and visiting other branches. DFS is often used to check graph connectivity, check cycles, and find paths. The time complexity of DFS is $O(V+E)$, với V là số đỉnh và E là số cạnh của đồ thị

2.3.3 Astar:

The A* algorithm is an algorithm for finding the shortest path in a graph or grid. It combines the actual cost and the remaining cost estimate to prioritize the paths with the lowest total cost. A* is suitable for the problem

of finding paths in state space and is widely used in applications such as finding paths on maps, in games and in autonomous robots. The time and space complexity of A* is $O(b^d)$,where b is the average number of branches per node and d is the depth of the shortest path.

2.3.4 Dijkstra'algorithm:

Dijkstra's algorithm is used to find the shortest path from a starting vertex to all other vertices in a graph with positive weights. It traverses the vertices by width and does not select the traversed vertex to ensure the shortest cost. Dijkstra is often applied in problems related to networks and routing. Dijkstra's time and space complexity are relatively efficient, given the time complexity is $O((V+E)\log V)$ if using priority queue.

2.3.5 Greedy best first search algorithm:

The Greedy Best-First Search algorithm is a method for finding paths in graphs. It selects the next vertex based on the estimate of the cost to reach the destination without considering the actual cost. Greedy Best-First Search is suitable for state space search problems with destination cost estimates. However, finding the shortest path is not guaranteed and may lead to suboptimal results in some cases

3. IMPLEMENTATION PLAN

3.1 Plan

Week	Works	Begin	End	Result
9	Plan a topic	16/10/2023	22/10/2023	Complete
10-11	Code algorithms	23/10/2023	4/11/2023	Complete
12	Build GUIs	6/11/2023	12/11/2023	Complete
13	Complete code and fix errors	14/11/2023	20/11/2023	Complete
14	Make report	21/11/2023	26/12/2023	Complete
15	Submit and correct errors a second time (if any)	27/11/2023	3/12/2023	Complete

3.2 Task

Num	Student Name	Work Description	Contribution
1	Do Huynh Bao Dang (leader)	- Code BFS - Build GUIs - Create player class, game class - Create one player funcion - Make report and ptt	100%
2	Nguyen Duy Manh	- Code DFS, Astart - Create maze class - Create two bot, change map funcion	100%

		- Make report and ppt	
3	Do Xuan Truong	- Code Dijikstra , GBFS algorithms - Bulid GUIs - Create clock class - Create two player funcion - Make report and ppt	100%

4. IDEA, INTERFACE, ALGORITHM OF THE PROBLEM.

4.1 Idea

4.2 GUI

4.2.1 Create Map

- Create map by a excel.csv with the 4 directions are east, west, south, and north with the number 1 representing the path and 0 representing the wall. Cell representing the Coordinate (row and column)

	A	B	C	D	E
1	cell	right	left	top	bottom
2	(1, 1)	1	0	0	1
3	(2, 1)	0	0	1	1
4	(3, 1)	1	0	1	1
5	(4, 1)	1	0	1	1
6	(5, 1)	1	0	1	0
7	(6, 1)	1	0	0	1
8	(7, 1)	1	0	1	1
9	(8, 1)	0	0	1	1
10	(9, 1)	1	0	1	0
11	(10, 1)	1	0	0	1
12	(11, 1)	1	0	1	1
13	(12, 1)	1	0	1	1
14	(13, 1)	1	0	1	1
15	(14, 1)	0	0	1	1
16	(15, 1)	1	0	1	1
17	(16, 1)	1	0	1	1

Figure 1: excel.csv

- And this is the code that can read the data from excel.csv

```
def upload_map(self, loadMaze):
        with open(loadMaze, 'r') as f:
            last = list(f.readlines())[-1]
            c = last.split(',')
            c[0] = int(c[0].lstrip('"('))
            c[1] = int(c[1].rstrip(')"'))
            self.rows = c[0]
            self.cols = c[1]
```

```
            # self.grid = []
        with open(loadMaze, 'r') as f:
            r = csv.reader(f)
            next(r)
            for i in r:
                c = i[0].split(',')
                c[0] = int(c[0].lstrip('('))
                c[1] = int(c[1].rstrip(')'))

                self.maze_map.append({'position': tuple(c),'right': int(i[1]), 'left': int(i[2]), 'top': int(i[3]), 'bottom': int(i[4])})
                self.maze_map_run[tuple(c)]={'right':int(i[1]),'left':int(i[2]),'top':int(i[3]),'bottom':int(i[4])}
def draw_walls(self, screen, cell_size):
        wall_color = (0, 0, 0)
        path_color = (255, 0, 0)   # Color for the start position
        start_position = (10, 10)   # Set the start position here

        for cell_data in self.maze_map:
            position = cell_data['position']
            x, y = position[1] * cell_size, position[0] * cell_size
            if cell_data['top'] == 0:
                pygame.draw.line(screen, wall_color, (x, y), (x + cell_size, y), self.thickness)
            if cell_data['right'] == 0:
                pygame.draw.line(screen, wall_color, (x + cell_size, y), (x + cell_size, y + cell_size), self.thickness)
            if cell_data['bottom'] == 0:
                pygame.draw.line(screen, wall_color, (x, y + cell_size), (x + cell_size, y + cell_size), self.thickness)
            if cell_data['left'] == 0:
                pygame.draw.line(screen, wall_color, (x, y), (x, y + cell_size), self.thickness)
            if position == start_position:
                pygame.draw.rect(screen, path_color, (x, y, cell_size, cell_size), self.thickness)

        if start_position == self.get_last_position():
                pygame.draw.rect(screen, path_color, (x, y, cell_size, cell_size), self.thickness)
```

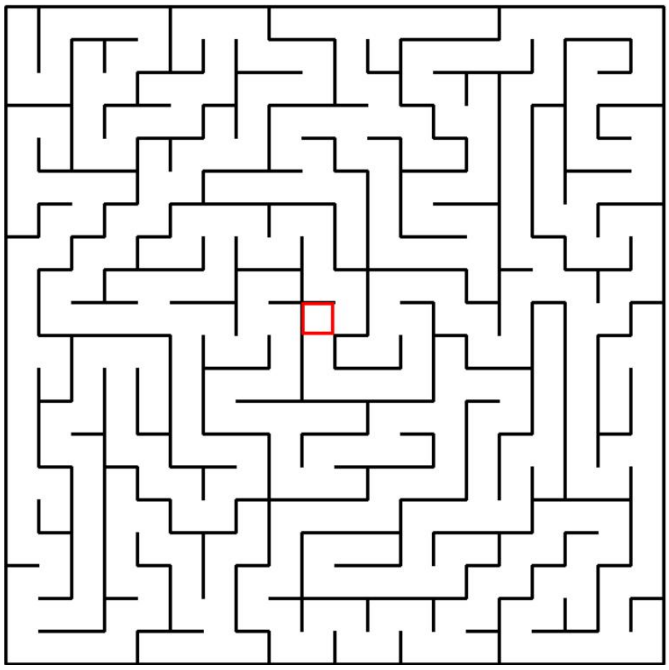

Figure 2: Interface for maze

4.2.2 Create Map

Just simple like a square with the color that we can set it

```
self.player_size = 30
self.rect = pygame.Rect(self.x, self.y, self.player_size, self.player_size)
```

```
def draw(self, screen, color):
    pygame.draw.rect(screen, color , self.rect)
```

4.2.3 Create a goal position

- Since the maze has 4 goals, we randomly chose to make it more interesting with 4 gates arranged in the 4 corners of the matrix.

```
def get_goal_position_random(self):
    last_position = None   # Initialize with a default value
    r = randint(1, 4)   # Fix: Use randint instead of random and correct the range
    if r == 1:
        last_position = (1, 1)
    elif r == 2:
```

```
            last_position = (1, 20)
        elif r == 3:
            last_position = (20, 1)
        elif r == 4:
            last_position = (20, 20)
        return last_position
```

4.3 Algorithm

4.3.1 BFS Algorithm

4.3.1.1 Idea

With unweighted graph and source vertex s. This graph can be directed or undirected, it does not matter to the algorithm.

The algorithm can be understood as a fire spreading on the graph: At step 0, only source vertex s is burning.

At each subsequent step, the fire burning at each vertex spreads to all adjacent vertices.

In each iteration of the algorithm, the "ring of fire" spreads out in width. The peaks closer to s will burn up first.

More precisely, the algorithm can be described as follows:

First we visit the source vertex s.

Visiting vertex s will generate the order of visiting vertices (u1, u2,...up) adjacent to s (the vertices closest to s). Next, we visit vertex u1. When we visit vertex u1, we will again be asked to visit vertices (v1, v2, ..., vp) adjacent to u1. But clearly these v vertices are "further" from s than uu vertices, so they can only be visited when all uu vertices have been visited. That is, the order of visiting vertices will be: s, u1, u2, …, up, v1, v2, …, vp.

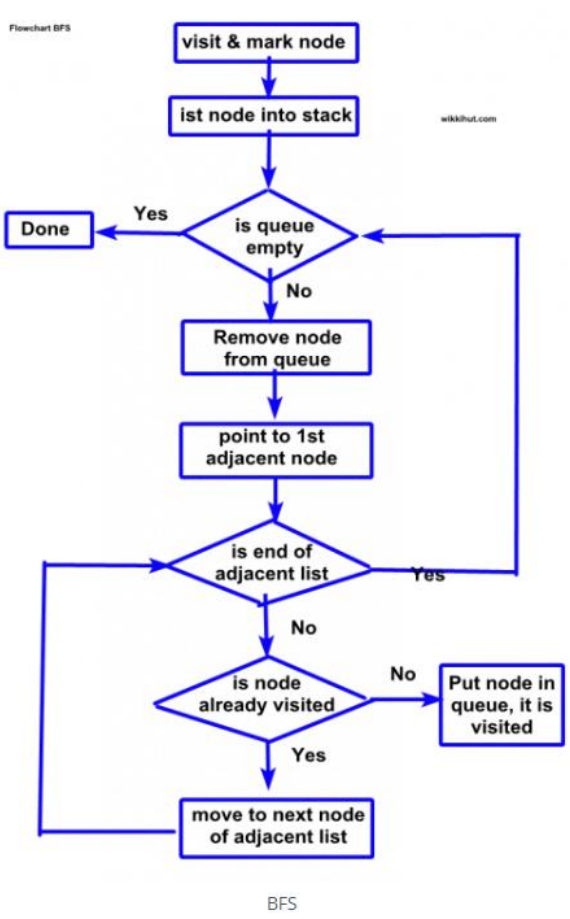

Figure 3: Flowchart of BFS algorithm

4.3.1.2 Description in Python

```
from maze import Maze
import pygame
import time
from collections import deque


class BFSSolver:


    def BFS(m, goal_position):
        start = (10, 10)
        # start = goal_position


        frontier = deque()
        frontier.append(start)
        bfsPath = {}
        visited_nodes = 0
```

```
explored = [start]
bSearch = []

start_time = time.time()

while len(frontier) > 0:
    currCell = frontier.popleft()
    visited_nodes += 1
    if currCell == goal_position:
        break
    for d in ['right', 'bottom', 'top', 'left']:
        if m.maze_map_run[currCell][d] == True:
            if d == 'right':
                childCell = (currCell[0], currCell[1] + 1)
            elif d == 'left':
                childCell = (currCell[0], currCell[1] - 1)
            elif d == 'top':
                childCell = (currCell[0] - 1, currCell[1])
            elif d == 'bottom':
                childCell = (currCell[0] + 1, currCell[1])
            if childCell in explored:
                continue
            frontier.append(childCell)
            explored.append(childCell)
            bfsPath[childCell] = currCell
            bSearch.append(childCell)
end_time = time.time()
execution_time = end_time - start_time

fwdPath = {}
cell = goal_position

fwdPath[bfsPath[cell]] = cell

while cell != (10, 10):
    fwdPath[bfsPath[cell]] = cell
    cell = bfsPath[cell]
```

```
    result_list = list(fwdPath.values())
    result_list = list(reversed(result_list))


    return result_list, fwdPath, bSearch, visited_nodes, execution_time
```

4.3.2 DFS Algorithm

4.3.2.1 Idea

Depth-First Search (DFS) is an algorithm that explores a graph by going as deep as possible along each branch before backtracking. It traverses a graph or tree in a depthward motion and uses a stack to remember which vertices to visit next. DFS does not use a heuristic evaluation; instead, it systematically explores the graph without considering the distances or costs associated with the edges.

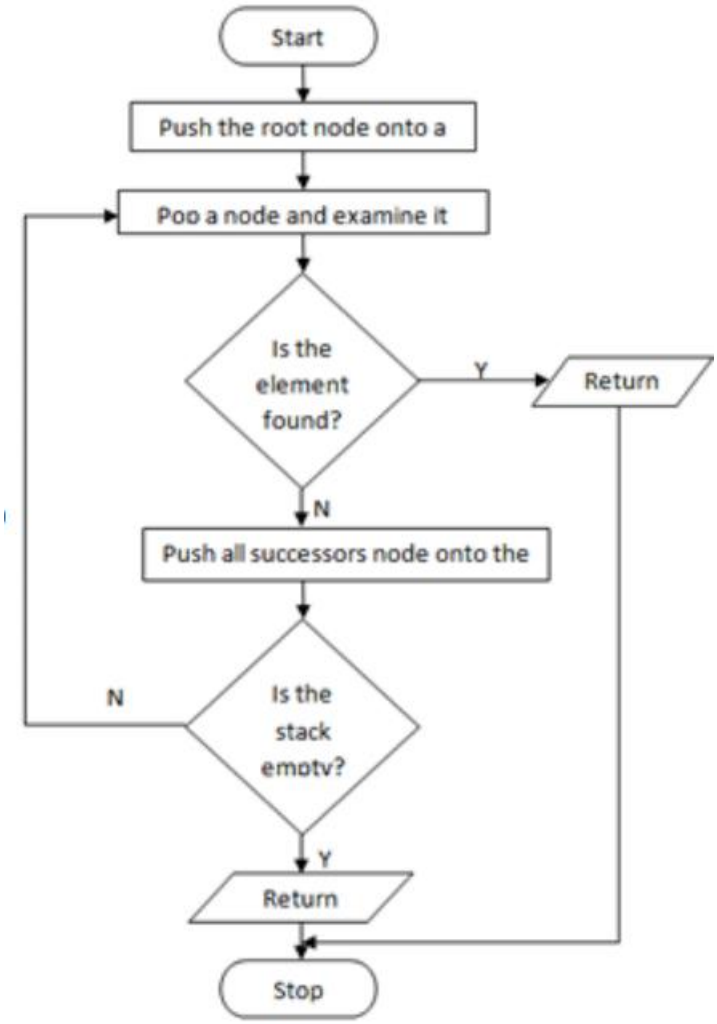

Figure 4: Flowchart of BFS algorithm

4.3.2.2 Description in Python

DFS starts at the initial node and explores as far as possible along each branch before backtracking. It can be implemented using recursion or an explicit stack.

```
from queue import LifoQueue
import pygame
from maze import Maze
import time

class DFSSolver:
    def DFS(m,goal_position):
        start=(10,10)
        explored=[start]
        frontier=[start]
        dfsPath={}
        dSeacrh=[]
        visited_nodes =0
        start_time = time.time()

        while len(frontier)>0:
            currCell=frontier.pop()
            visited_nodes += 1
            dSeacrh.append(currCell)
            if currCell==goal_position:
                break
            poss=0
            for d in ['right', 'left', 'top', 'bottom']:
                if m.maze_map_run[currCell][d] == 1:
                    if d == 'right':
                        child = (currCell[0], currCell[1] + 1)
                    elif d == 'left':
                        child = (currCell[0], currCell[1] - 1)
                    elif d == 'top':
                        child = (currCell[0] - 1, currCell[1])
                    elif d == 'bottom':
```

```
                    child = (currCell[0] + 1, currCell[1])
                if child in explored:
                    continue

                explored.append(child)
                frontier.append(child)
                dfsPath[child]=currCell

    end_time = time.time()
    execution_time = end_time - start_time

    fwdPath={}
    cell=goal_position
    while cell!=start:
        fwdPath[dfsPath[cell]]=cell
        cell=dfsPath[cell]
    result_list = list(fwdPath.values())
    result_list = list(reversed(result_list))

    return result_list,fwdPath,dSeacrh, visited_nodes, execution_time
```

4.3.3 A* Algorithm

4.3.3.1 Idea

This algorithm finds a path from a start node to a given destination node (or to a node that satisfies a destination condition). This algorithm uses a "heuristic evaluation" to rank each node according to its estimate of the best route through that node. This algorithm traverses the nodes in the order of this heuristic evaluation. Therefore, the A* algorithm is an example of best-first search.

4.3.3.2 Description in Python

A* stores a set of incomplete solutions, that is, paths through the graph, starting from the starting node. This solution set is stored in a priority queue. The priority order is decided by the Heuristic function f(x) = g(x) +h(x)

In this problem, h(x) is the distance from the current node to the destination node

g(x) is the distance calculated as the crow flies from the current node to the node to be considered f(x) = g(x)+ h(x)

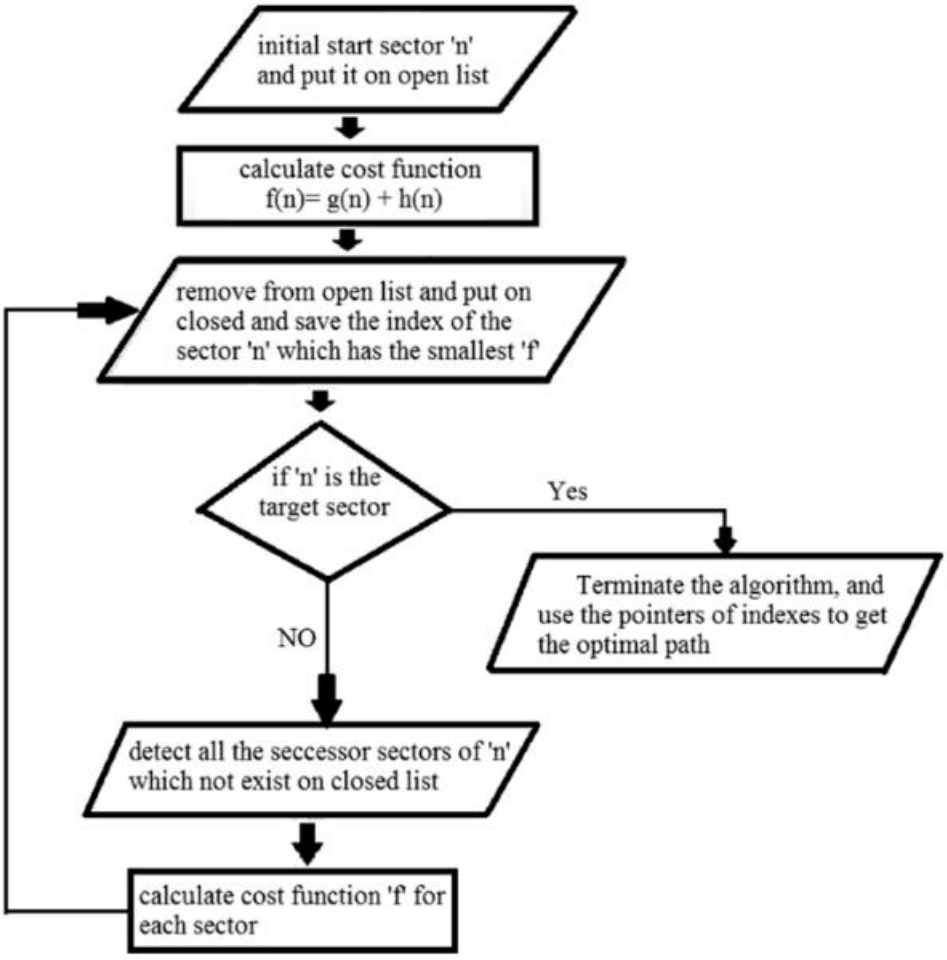

Figure 5: Flowchart of BFS algorithm

```
from queue import PriorityQueue
import pygame
from maze import Maze
import time


class AStarSolver:
    def h(cell1, cell2):
        x1, y1 = cell1
        x2, y2 = cell2
```

```
        return abs(x1 - x2) + abs(y1 - y2)

    def aStar(m,goal_position):
        # start = (10,10)
        start = (10,10)
        open = PriorityQueue()
        open.put((AStarSolver.h(start, goal_position), AStarSolver.h(start, goal_position), start))
        aPath = {}
        g_score = {cell: float('inf') for cell in m.maze_map_run}
        g_score[start] = 0
        f_score = {cell: float('inf') for cell in m.maze_map_run}
        f_score[start] = AStarSolver.h(start, goal_position)
        searchPath=[start]
        visited_nodes = 0

        start_time = time.time()

        while not open.empty():
            currCell = open.get()[2]
            visited_nodes += 1    # Increment the visited nodes counter
            searchPath.append(currCell)
            if currCell == goal_position:
                break
            for d in ['right', 'bottom', 'top', 'left']:
                if m.maze_map_run[currCell][d]==True:
                    if d=='right':
                        childCell=(currCell[0],currCell[1]+1)
                    elif d=='left':
                        childCell=(currCell[0],currCell[1]-1)
                    elif d=='top':
                        childCell=(currCell[0]-1,currCell[1])
                    elif d=='bottom':
                        childCell=(currCell[0]+1,currCell[1])

                    temp_g_score = g_score[currCell] + 1
                    temp_f_score = temp_g_score + AStarSolver.h(childCell, goal_position)

                    if temp_f_score < f_score[childCell]:
                        g_score[childCell] = temp_g_score
```

```
                    f_score[childCell] = temp_g_score + AStarSolver.h(childCell, goal_position)
                    open.put((f_score[childCell], AStarSolver.h(childCell, goal_position), childCell))
                    aPath[childCell] = currCell

    end_time = time.time()
    execution_time = end_time - start_time

    fwdPath={}
    cell=goal_position
    while cell!=start:
        fwdPath[aPath[cell]]=cell
        cell=aPath[cell]
    result_list = list(fwdPath.values())
    result_list = list(reversed(result_list))
    print("Visited Nodes:", visited_nodes)   # Print the visited nodes count

    return result_list,fwdPath,searchPath,visited_nodes, execution_time
```

4.3.4 Greedy Best-First Search Algorithm

4.3.4.1 Idea

The GBFS algorithm will evaluate the vertices using the heuristic function h(n), meaning the evaluation function will be equal to the heuristic function f(n) = h(n). This makes the algorithm ‘greedy’.

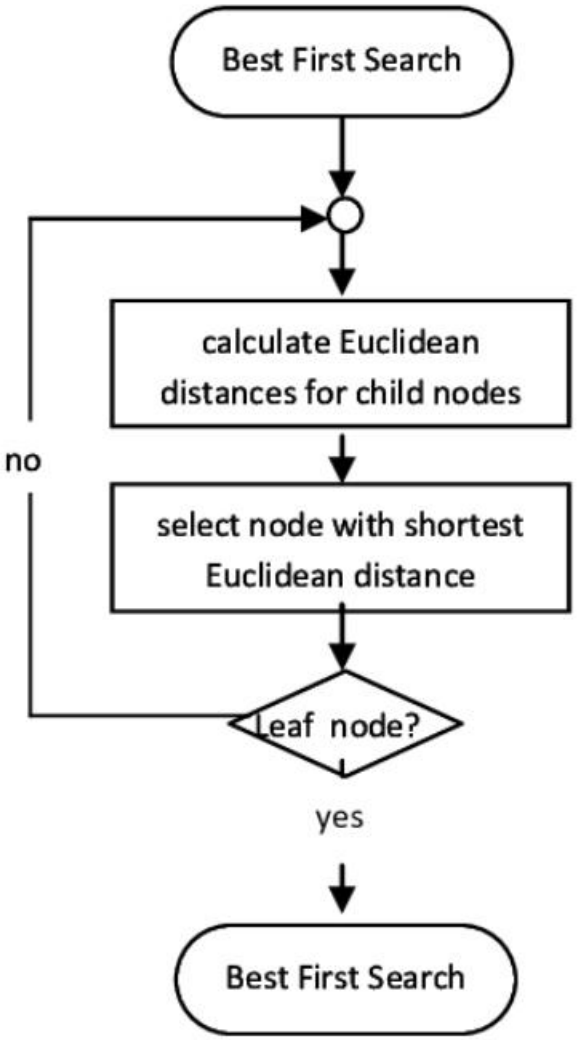

Figure 6: Flowchart of BFS algorithm

4.3.4.2 Description in Python

The Greedy Best-First Search algorithm will use an evaluation function, the heuristic function h(n).

This heuristic function h(n) will evaluate the cost to go from current node n to destination node

```
from queue import PriorityQueue
import pygame
```

```
from maze import Maze
import time

class GreedySolver:
    def h(cell, goal_position):
        x, y = cell
        goal_x, goal_y = goal_position
        return abs(x - goal_x) + abs(y - goal_y)

    def greedy(m, goal_position):
        start = (10,10)
        open = PriorityQueue()
        open.put((GreedySolver.h(start, goal_position), start))
        aPath = {}
        searchPath = [start]
        visited_nodes = 0

        start_time = time.time()

        while not open.empty():
            _, currCell = open.get()
            visited_nodes += 1   # Increment the visited nodes counter
            searchPath.append(currCell)

            if currCell == goal_position:
                break

            for d in ['right', 'bottom', 'top', 'left']:
                if m.maze_map_run[currCell][d]:
                    if d == 'right':
                        childCell = (currCell[0], currCell[1] + 1)
                    elif d == 'left':
                        childCell = (currCell[0], currCell[1] - 1)
                    elif d == 'top':
                        childCell = (currCell[0] - 1, currCell[1])
                    elif d == 'bottom':
                        childCell = (currCell[0] + 1, currCell[1])

                    if childCell not in aPath:
```

```
                    open.put((GreedySolver.h(childCell, goal_position), childCell))
                    aPath[childCell] = currCell
    end_time = time.time()
    execution_time = end_time - start_time

    fwdPath = {}
    cell = goal_position
    while cell != start:
        fwdPath[aPath[cell]] = cell
        cell = aPath[cell]
    result_list = list(fwdPath.values())
    result_list = list(reversed(result_list))

    return result_list, fwdPath, searchPath, visited_nodes, execution_time
```

4.3.5 Dijkstra's Algorithm

4.3.5.1 Idea

Dijkstra's algorithm is used to find the shortest path from a source vertex s to all other vertices in a graph with non-negative edge weights. It maintains a set of vertices whose shortest distance from the source is known and continually expands this set until all vertices are included.

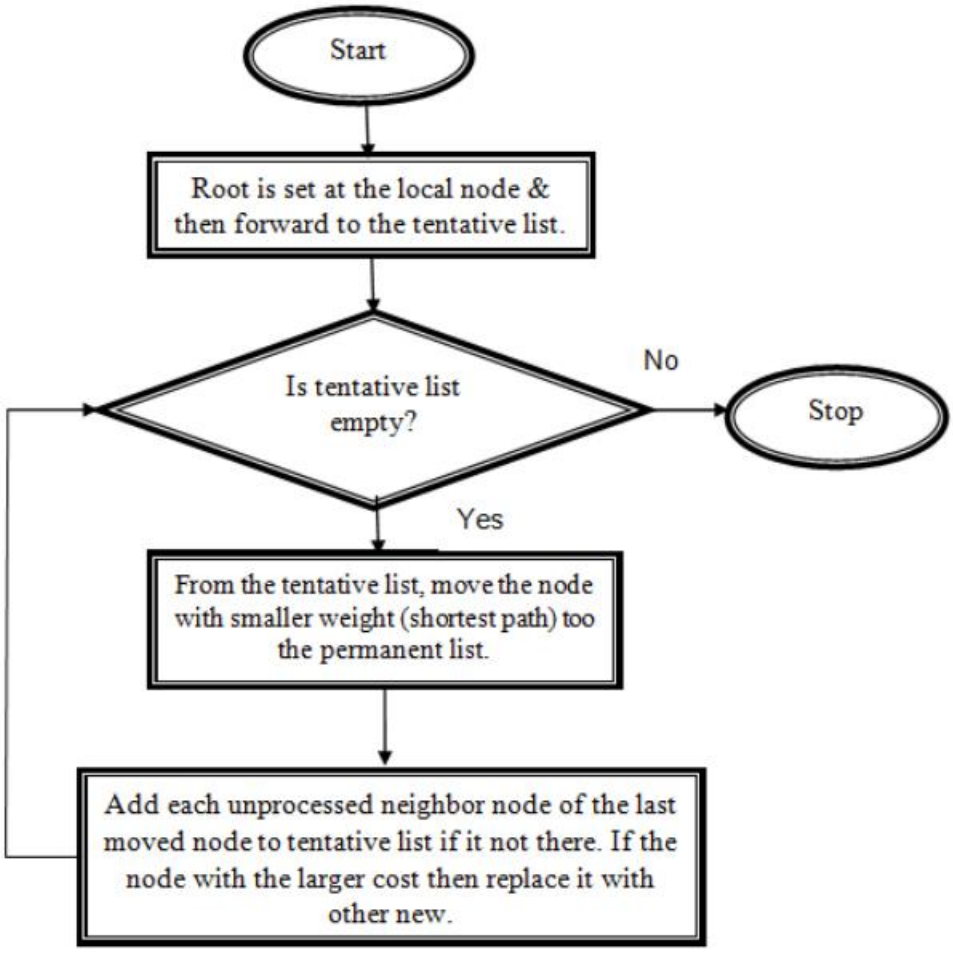

Figure 7: Flowchart of Dijkstra algorithm

4.3.5.2 Description in Python

```
from queue import PriorityQueue
import pygame
from maze import Maze
import time


class DijkstraSolver:
    def dijkstra(m,goal_position):
        start =(10,10)
        open = PriorityQueue()
        open.put((0, start))
        dist = {cell: float('inf') for cell in m.maze_map_run}
        dist[start] = 0
```

```
aPath = {}
searchPath = [start]
visited_nodes = 0   # Counter for visited nodes

start_time = time.time()

while not open.empty():
    currCost, currCell = open.get()
    visited_nodes += 1   # Increment the visited nodes counter
    searchPath.append(currCell)

    if currCell == goal_position:
        break

    for d in ['right', 'bottom', 'top', 'left']:
        if m.maze_map_run[currCell][d]:
            if d == 'right':
                childCell = (currCell[0], currCell[1] + 1)
            elif d == 'left':
                childCell = (currCell[0], currCell[1] - 1)
            elif d == 'top':
                childCell = (currCell[0] - 1, currCell[1])
            elif d == 'bottom':
                childCell = (currCell[0] + 1, currCell[1])

            edgeCost = 1   # Assuming each step has a cost of 1

            if dist[currCell] + edgeCost < dist[childCell]:
                dist[childCell] = dist[currCell] + edgeCost
                open.put((dist[childCell], childCell))
                aPath[childCell] = currCell
end_time = time.time()
execution_time = end_time - start_time

fwdPath = {}
cell = goal_position
while cell != start:
    fwdPath[aPath[cell]] = cell
    cell = aPath[cell]
```

```
    result_list = list(fwdPath.values())
    result_list = list(reversed(result_list))
    # print("Visited Nodes:", visited_nodes)   # Print the visited nodes count


    return result_list, fwdPath, searchPath,visited_nodes, execution_time
```

5. FUNCTION.

5.1 Player function

```
def check_move(self, tile,maze_map_run ,thickness):
        current_cell_x, current_cell_y = self.y // tile, self.x // tile
        current_cell_abs_x, current_cell_abs_y = current_cell_y * tile, current_cell_x * tile

        print("Player:", self.y, self.x)
        print("Current Cell:", current_cell_x, current_cell_y)

        if self.left_pressed:
            print("Checking Left:", maze_map_run[current_cell_x, current_cell_y]['left'])

            if maze_map_run[current_cell_x,current_cell_y]['left'] == 0:
                    self.left_pressed = False
        if self.right_pressed:
            print("Checking Right:", maze_map_run[current_cell_x, current_cell_y]['right'])

            if maze_map_run[current_cell_x,current_cell_y]['right'] == 0:
                    self.right_pressed = False
        if self.up_pressed:
            print("Checking Top:", maze_map_run[current_cell_x, current_cell_y]['top'])

            if maze_map_run[current_cell_x,current_cell_y]['top'] == 0:
                    self.up_pressed = False
        if self.down_pressed:
            print("Checking Bottom:", maze_map_run[current_cell_x, current_cell_y]['bottom'])

            if maze_map_run[current_cell_x,current_cell_y]['bottom'] == 0:
                    self.down_pressed = False
```

5.2 Bot function

```
    if 800 <= mouse_x <= 800 + 100 and \
            170 <= mouse_y <= 170 + 50:
        clock.start_timer()
        player = Player(30*10,30*10)
        player1 = Player(30*10,30*10)
        dfs_path, dfs_fwdPath, dfsSearch, visited_nodes, execution_time   = DFSSolver.DFS(maze, self.goal_position)
```

```
        font = pygame.font.Font(None, 36)
        text1 = font.render(f'Visited Nodes: {visited_nodes}', True, (0, 0, 0))
        screen.blit(text1, (50,650))
        text2 = font.render(f'Time: {execution_time*1000}', True, (0, 0, 0))
        screen.blit(text2, (50,670))

           for step in dfsSearch:
                player1.follow_searchPath([step], self.screen)
                self._draw(maze, tile, player1, game, clock,self.color_blue)
                pygame.time.delay(10)
                self.screen.fill(pygame.Color("darkslategray"), (650, 0, 752, 752))

          for step in dfs_fwdPath:
                 player.follow_searchPath([step], self.screen)
                  self._draw(maze, tile, player, game, clock,self.color_green)
                  pygame.time.delay(100)
                  self.screen.fill(pygame.Color("darkslategray"), (650, 0, 752, 752))

          for step in dfs_path:
                  if game.is_game_over(player,self.goal_position):
                       self.game_over = True
                  else:
                       self.game_over = False
                       player.follow_searchPath([step], self.screen)
                       self._draw(maze, tile, player, game, clock,self.color_red)
                       pygame.time.delay(100)
                       self.screen.fill(pygame.Color("darkslategray"), (650, 0, 752, 752))
                  self.game_over = True

           print('DFS')
```

5.3 Bot vs Bot

```
if 660 <= mouse_x <= 660 + 100 and \
        460 <= mouse_y <= 460 + 50:
    clock.start_timer()
    self.goal_position2 = self.get_goal_position_random()
```

```
    while (self.goal_position2 == self.goal_position):
        self.goal_position2 = self.get_goal_position_random()
    player2 = Player(30*10,30*10)
    greedy_path, greedy_fwdPath, GreedySearch,visited_nodes, execution_time   = GreedySolver.greedy(maze, self.goal_position)
    greedy_path2, greedy_fwdPath2, GreedySearch2,visited_nodes2, execution_time2 = GreedySolver.greedy(maze, self.goal_position2)
    for step, step2 in zip(greedy_path, greedy_path2):
        player.follow_path([step], self.screen)
        player2.follow_path([step2], self.screen)
        self._drawTwoPlayer(maze, tile, player,player2, game, clock,self.color_green,self.color_blue)
        pygame.time.delay(200)

        self.screen.fill(pygame.Color("darkslategray"), (650, 0, 752, 752))

        if game.is_game_over(player,self.goal_position) :
            self.game_over = True
        elif game.is_game_over(player2,self.goal_position2) :
            self.winner = 1
            self.game_over = True
        else:
            self.game_over = False
print('Two bot')
```

5.4 Player vs Player

```
def _drawTwoPlayer(self, maze, tile, player,player2, game, clock, color, color2):

        # draw maze
        maze.draw_walls(screen, tile)
        # add a goal point to reach1111
        game.add_goal_point(self.screen,self.goal_position)
        if(self.goal_position2 != None):
            game.add_goal_point(self.screen,self.goal_position2)

        # draw every player movement
        player.draw(self.screen,color)
        player.update()

        player2.draw(self.screen,color2)
```

```
        player2.update()

        # instructions, clock, winning message
        self.instructions()
        if self.game_over:
            if self.winner == 0:
                clock.stop_timer()
                self.screen.blit(game.message(),(660,520))
            elif self.winner == 1:
                clock.stop_timer()
                self.screen.blit(game.message2(),(660,520))
        else:
            clock.update_timer()
        self.screen.blit(clock.display_timer(), (710,550))

        pygame.display.flip()
```

```
if 660 <= mouse_x <= 660 + 100 and \
                    400 <= mouse_y <= 400 + 50:
                clock.start_timer()
                player2 = Player(30*11,30*11)
                print('Two player')
```

```
if(player2 == None):
                self._draw(maze,tile,player,game,clock,self.color_green)
            else:
                self._drawTwoPlayer(maze ,tile,player,player2 ,game, clock, self.color_green, self.color_blue)
```

6. EXPERIMENT, ANALYZE, EVALUATE RESULTS.

6.1 Interface information

6.1.1. Interface when starting the game

- When we start the game, the interface will appear like Figure 1 to start the game.

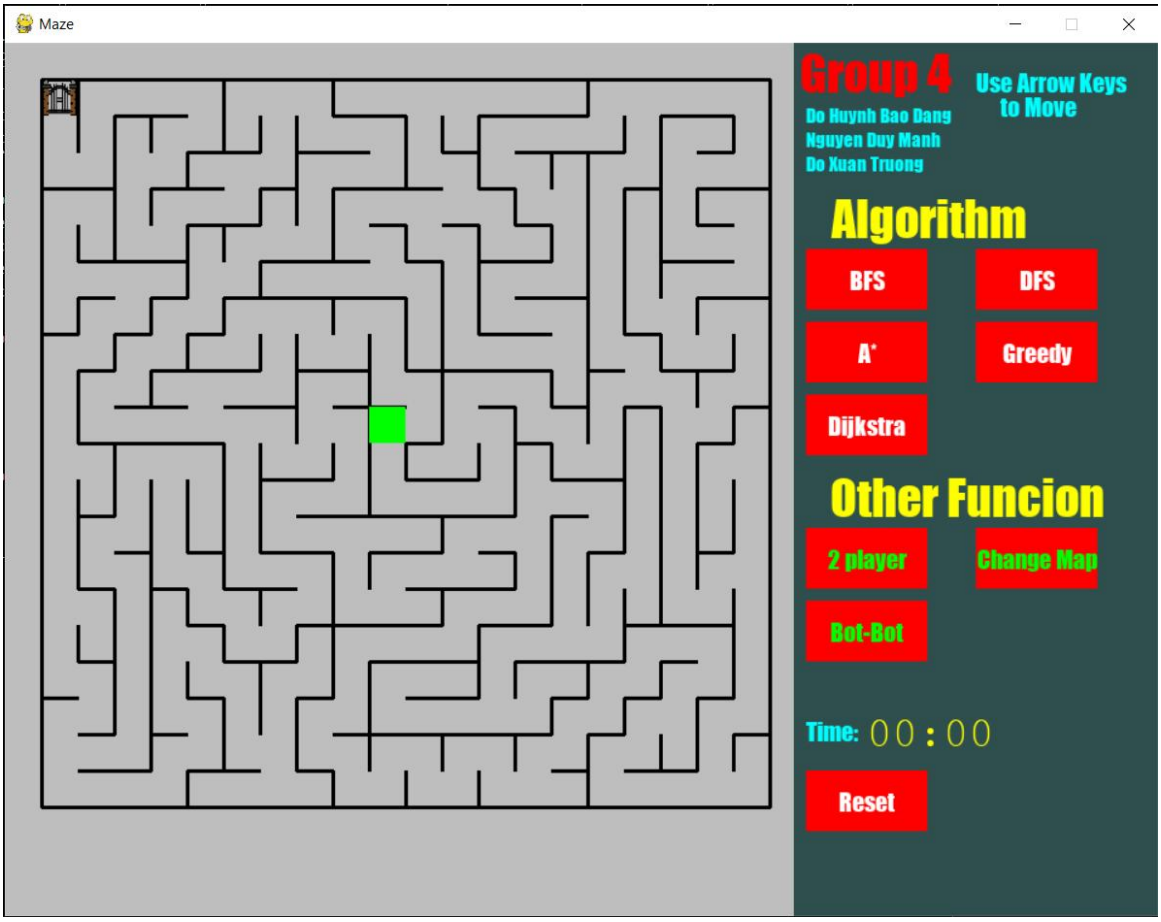

Figure 8: Interface when starting the game

6.1.2. Interface 4 goals and 3 mazes

- To change the goal, before starting the game we can select "Reset". Here we have 4 goals for the bot to reach

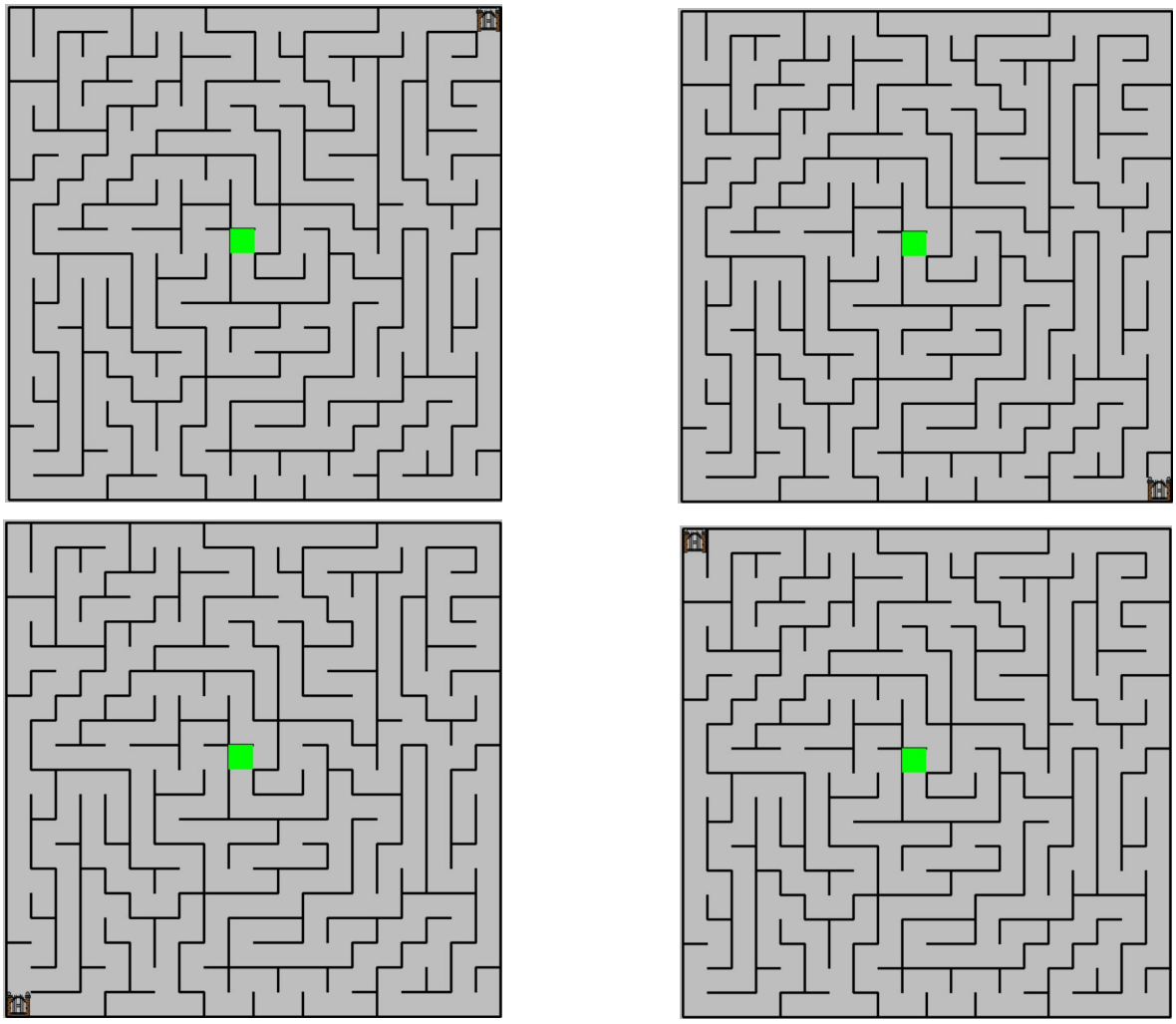

Figure 9: Interface for 4 goals

- To change the maze, before starting the game we can select "Change Map". Here we have 3 mazes for the user can choose.

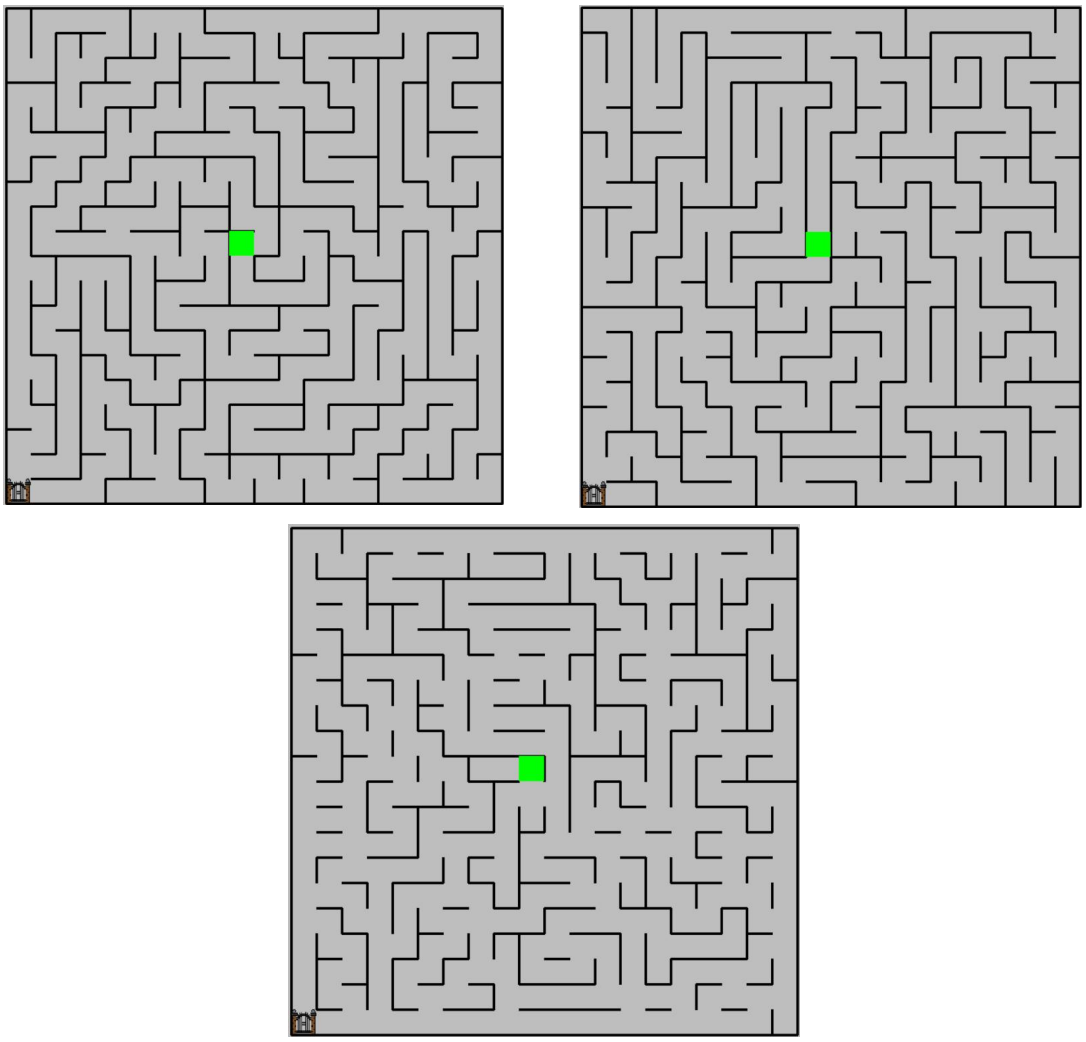

Figure 10: Interface for 3 mazes

6.1.3. Start Node (Player)

- To control the starting node, we use keycodes (W, S, A, D, UpArrow, DownArrow, LeftArrow, RightArrow)

- W, UpArrow to move the character to the top of

- S, DownArrow to move the character to the bottom of

- A, LeftArrow to move the character to the left

- D, RightArrow to move the character to the right

6.1.4. Start Node (Bot)

- For the bot to start running, we can click directly on the algorithms available on the game menu:

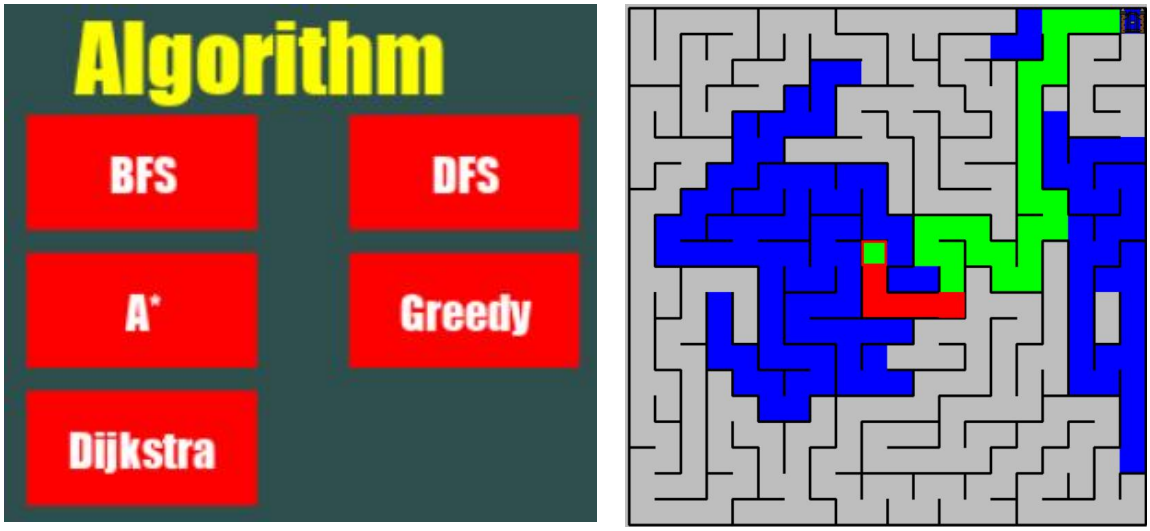

Figure 11: Interface for running bot

6.1.5. Game mode

- We have 2 game modes for 2:

+ Player vs player: In this mode, two players will use the keyboard to move and reach a destination, whoever comes first will be the winner.

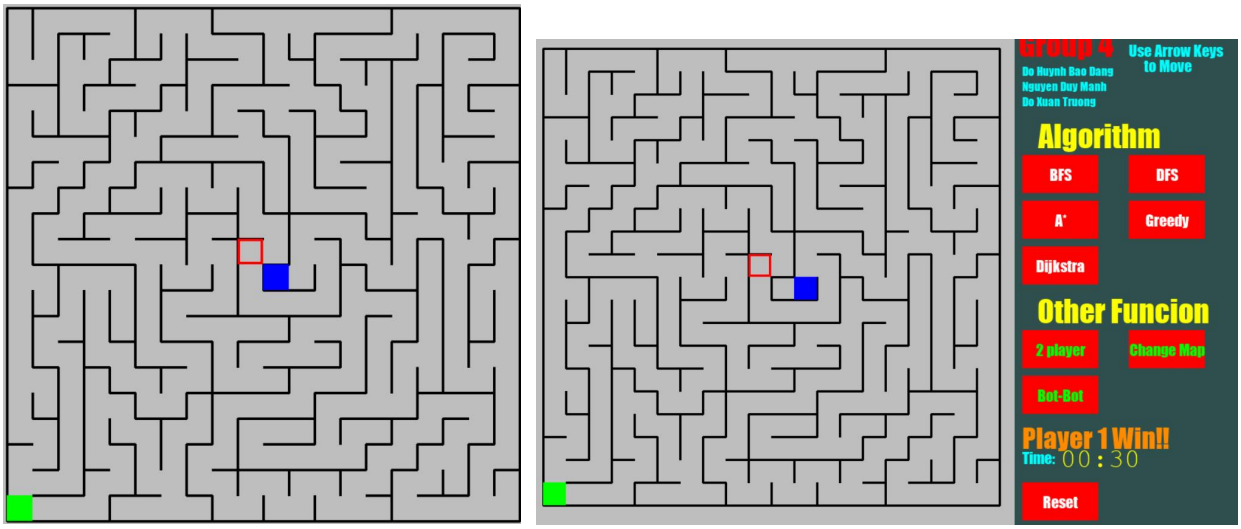

Figure 12: Interface for 2 players

+ Bot vs bot: In this mode, there will be a little difference, two bots will move to any 2 different gates, if the bot reaches the gate the fastest, it will win.

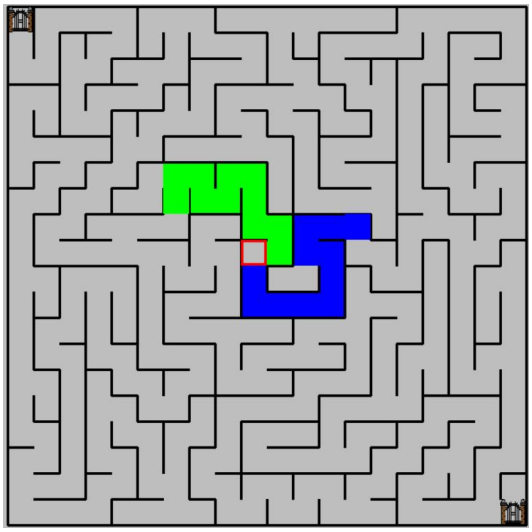

Figure 13: Interface for two bots

6.2. Analyze, experiment, evaluate

To distinguish between algorithms, find the most optimal solution using an algorithm based on different cases of a matrix. Here we use the same matrix in applying the 5 algorithms and create a summary table to compare and draw conclusions about the above algorithms.

6.2.1. Analyze, experiment

6.2.1.1. Maze one.

a, BFS

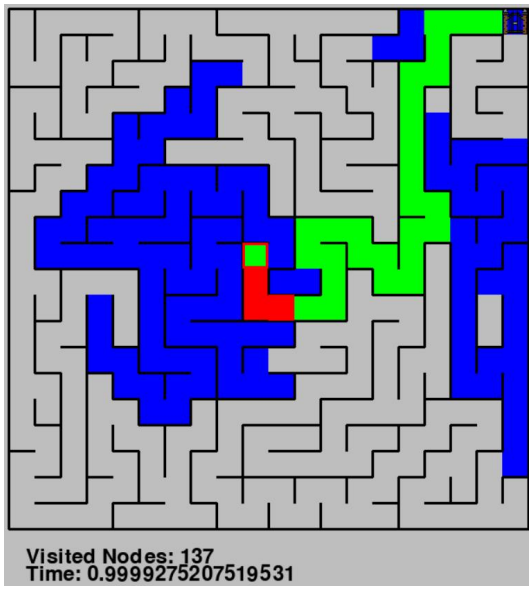

Figure 14: Interface for BFS

b, DFS

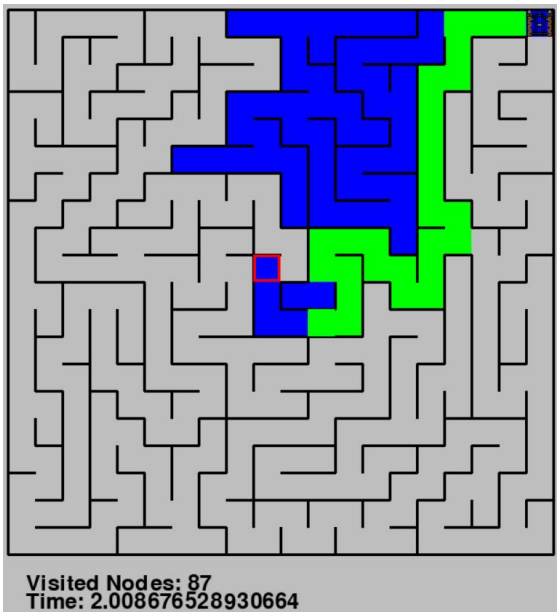

Figure 15: Interface for DFS

c, A*

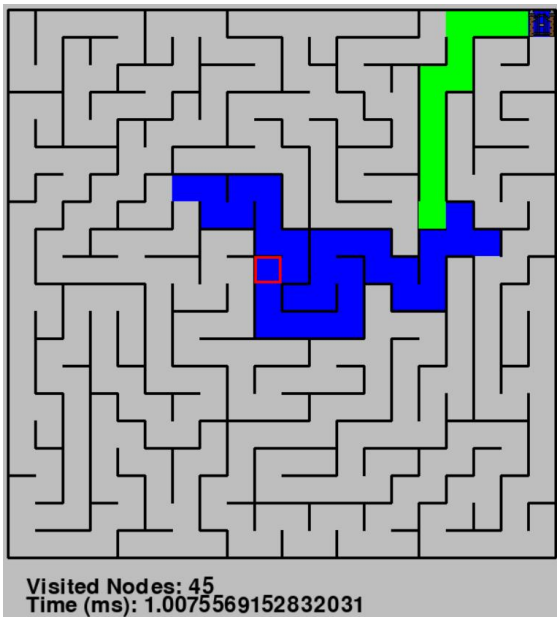

Figure 16: Interface for A Star

d, GBFS

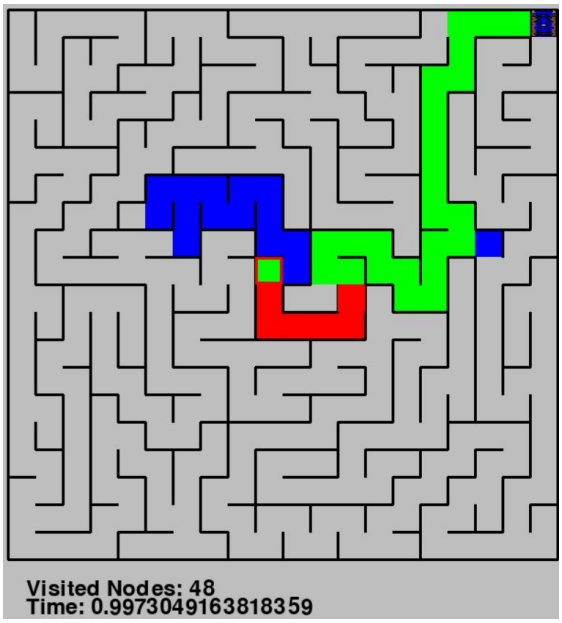

Figure 17: Interface for GBFS

e, Dijkistra's

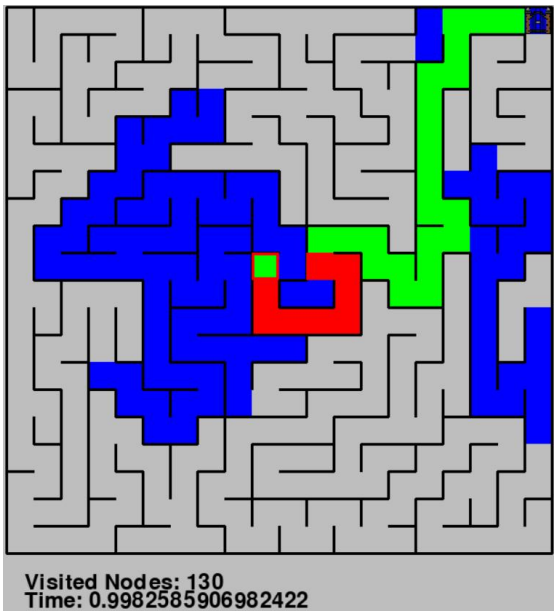

Figure 18: Interface for Dijkistra

6.3. Summary table

Statistical value Algorithm	Visited Node	Execution Time
BFS	137	0.999 ms

DFS	87	2.008 ms
A Star	45	1.008 ms
GBFS	48	0.997 ms
Dijkistra	130	0.998 ms

From the experimental analysis table obtained, we see:

- BFS (Breadth-First Search):

Visited Nodes: 137 - Out of the number of visited nodes, BFS can often visit more nodes than other algorithms, especially on large graphs or complex mazes.

Execution Time: 0.999 ms - Execution time is relatively fast, but sometimes BFS can take significant time if the graph is too large.

- DFS (Depth-First Search):

Visited Node: 87 - DFS typically has a lower number of visited nodes than BFS. It can take advantage of its "deep first, then broad" nature.

Execution Time: 2,008 ms - Higher execution time compared to BFS. DFS can get stuck in deep paths and thus take longer to complete.

- A Star:

Visited Node: 45 - A Star typically helps reduce the number of visited nodes by combining actual costs and estimated costs (heuristics).

Execution Time: 1,008 ms - Relatively fast execution time. A Star is often effective when it has a good heuristic function.

- GBFS (Greedy Best-First Search):

Visited Node: 48 - GBFS uses a "greedy" strategy, focusing on heuristic cost estimates to decide scaling steps.

Execution Time: 0.997 ms - Relatively fast execution time. GBFS may choose a suboptimal path if the heuristic function is incorrect.

- Dijkstra:

Visited Node: 130 - Dijkstra visits a larger number of nodes than A Star, because it does not use heuristics.

Execution Time: 0.998 ms - Relatively fast execution time. Dijkstra guarantees finding the shortest path on a graph with non-negative weights.

7. CONCLUSION.

7.1 Assessments:

From analyzing, evaluating, and testing our group, we realized BFS is the foundation algorithm for the development of other algorithms such as AStar and GBFS . With a simple small map, algorithms can almost always find the shortest path. The most optimal in terms of time and distance is still GBFS and if the maze game is weighted and complex, AStar may be more optimal when applying its calculation function to find the algorithm.

7.2 Development:

We will develop many different game modes such as player vs computer, develop the user interface, add many new functions such as applying weights to maze games,...

7.3 Experience learned:

- Learn python language
- Algorithm selection depends on specific requirements
- Space and time complexity are important factors
- Advantages and limitations of each algorithm
- The importance of modeling
- Understand the goals and characteristics of the game

8. REFERENCES.

8.1. How to Make a Maze Game in Python - Python Code (thepythoncode.com)

8.2. A* Search Algorithm - GeeksforGeeks

8.3. Breadth First Search or BFS for a Graph - GeeksforGeeks

8.4. Depth First Search or DFS for a Graph - GeeksforGeeks

8.5. C / C++ Program for Dijkstra's shortest path algorithm | Greedy Algo-7 - GeeksforGeeks

8.6. Greedy Best first search algorithm - GeeksforGeeks