A blue and white logo

Description automatically generated

Artificial Intelligent

PROJECT 1: Hide and Seek

**Thành viên:**

* Trần Đăng Tuấn – 22127438
* Nguyễn Phước Sang – 22127362
* Cao Minh Quang – 22127353
* Ngô Tấn Tài – 22127371

# Introduction

**Welcome to our ProJect 1: Hide and Seek**

In this Project, we are going to Python to program this one. There are serveral libraries which help us to finish this project are:

* pygame: it helps us to create the interface or make the GUI
* os: it helps us to get access to the location of our file
* time: it helps us to create waiting time when prints out the matrixs for easily tracking
* heapq: it supports us with the A Star function
* random: it helps us to choose the movements for hidders or seekers

Beginning, it is our welcome interface, it looks like this:

A screenshot of a computer game

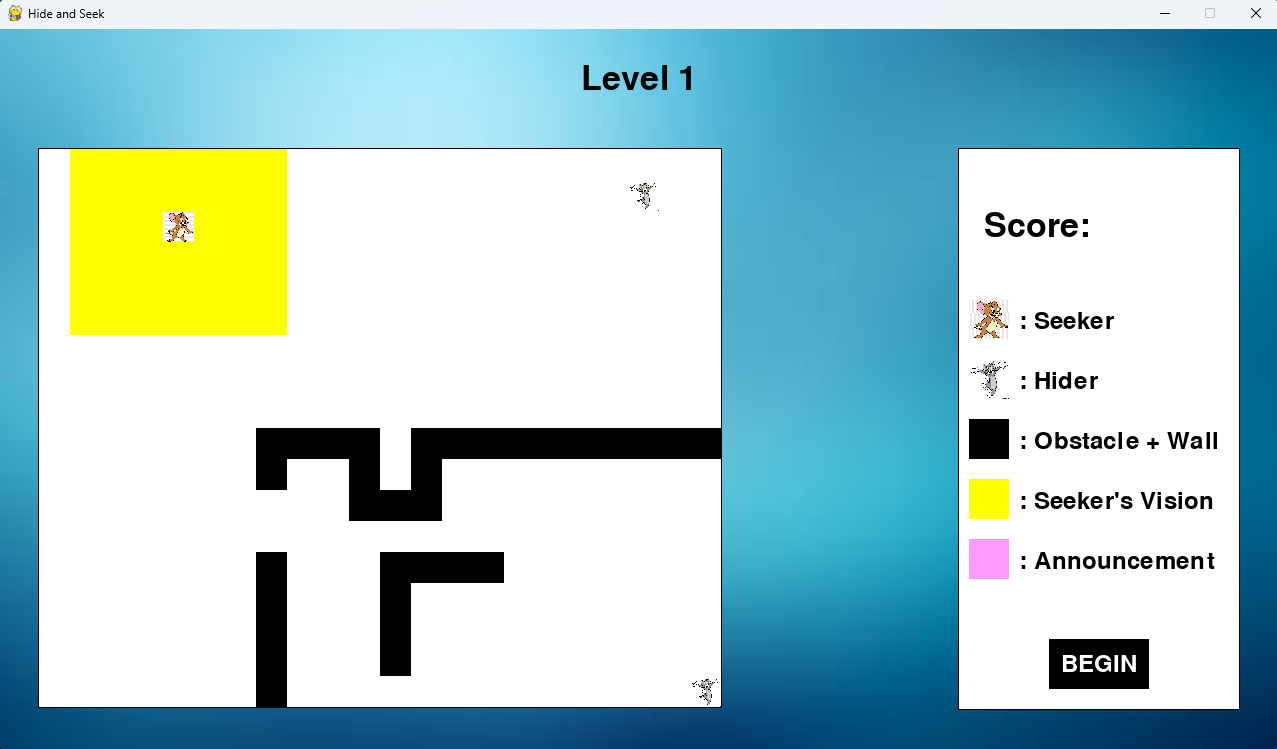
Description automatically generated

When you click on each level, it will appear the corresponding level. For example: I click on **Level 1**, it will appear the **Level 1 Screen**, with some choices of the map. Each level will at least 5 maps.

A screenshot of a computer

Description automatically generated

Then, you can choose the map you want to run our program. For example: I will click on **map1.txt**.



The white rectangle in the right side of the program is the legends, which show you the icon and its definition. Then when you click the button **Begin**, the seeker start to run to find the hider and calculate the score of the seeker.

# Assignment Plan

## First Step: Set up Schedule

Our team has made a work schedule in order to be easier for lecturers to keep track of how much time we spend on each sessions in our project.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | We  20 / 3 | Th | Fr | Sa | Su | Mo | Tu | We | Th | Fr | Sa | Su | Mo | Tu | We | Th | Fr | Sa | Su  07 / 4 |
| Level 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Level 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Level 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| GUI |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Report |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

## Second Step: Create essential Functions and Classes

For making GUI, we have created 2 classes:

class GUI:

class Map:

Each classes has the crucial role in our project. GUI, which will be interface, Map, which wil include the matrix, the important information about hiders, seeker.

There are also some algorithms such as Local Search and A star, that we use to run our program, we will go through the details of the algorithms in the next chapter.

In class GUI, there are many functions that supporting to create interface and read data such as:

def read\_matrix(self, file\_txt):

def draw\_button(self, x, y, text, WIDTH, HEIGHT, hover=False):

def draw\_matrix(self, matrix, HEIGHT, WIDTH, start\_x, start\_y, end\_x, end\_y):

def draw\_note(self, start\_x, start\_y, end\_x, end\_y):

def solve\_screen(self, map\_matrix : Map):

def level\_screen(self):

def run(self):

in class Map, there are also many functions and algorithms that we use during the project:

def getVision(self):

def checkHider(self, des):

def checkAnnouncement(self, des):

def findMostValueCell(self):

def moveSeeker(self):

def localSearch(self):

def A\_Star(self, goalRow, goalCol):

def chooseAnnouncePos(self, pos):

def findHider(self, hiderPos1):

def createAnnounce(self, step, tmpValue, tmpPos):

def moveHider (self, Pos):

We also create Map2, in order to solve the Level 3 problem with the same function as Map, but we change a little bit in these funtions. Finally, there are some supporting functions, such as:

def checkWall(row, col, board, maxRow, maxCol):

def checkUnValidCell(row, col, board, maxRow, maxCol):

def hideCell(row, col, board, maxRow, maxCol):

def findDiagonalDistance(cur\_row, cur\_col, goal\_row, goal\_col):

def findNumberOfWallAround(row, col, board, maxRow, maxCol):

def calcCellValue(cur\_row, cur\_col, goal\_row, goal\_col, board, ROW, COL):

def findManhattanDistance(cur\_row, cur\_col, goal\_row, goal\_col):

# Details for each Level

## Level 1 and Level 2

In these two level, our programing can be said to be similar to each Level. But when we finish programiing level 1, we still need to modify a little bit to be suitable with level 2.

First of all, we will read a file with the .txt type to input the matrix and display it on our interface. We have some values representing for some information like:

* 0: stands for an empty cell and that cell haven’t been visited before
* 1: stands for walls and obstacle: (though lv 1-2-3 agents don’t work with obstacle so we will represent it like wall)
* 2: stands for hider
* 3: stands for seeker
* -1: stands for a visited cell
* Plus 20 to a cell Value if that cell is in Seeker’s vision.

The next thing we do, is to call the function **getVision**, to show which cell can the seeker see. Let’s breakthrough the **getVision** function.

In **getVision**: first we will plus 20 to all the value of cells in the hider vision range (which is a 7x7 matrix that have center at the seeker) – which make all possiple cells (except wall) that can be seen by seeker, then we will check all cells arround seeker (8 cells arround seeker and 16 cell in the mid range – or a 5x5 matrix that have center at seeker), if that cell is a wall then we will hide all cells that will be blocked by decrease the value of that cell by 20 to give back the initial value. The rules for all cells that a wall can hide is applied by following the restrictions in the requirements.

        if (checkWall(row - 1, col - 1, self.board, self.row, self.col)):

            hideCell(row - 2, col - 2, self.board, self.row, self.col)

            hideCell(row - 3, col - 3, self.board, self.row, self.col)

            hideCell(row - 2, col - 3, self.board, self.row, self.col)

            hideCell(row - 3, col - 2, self.board, self.row, self.col)

After the **getVision** function, we will calculate the most value cell to run **A** **Star** to that position, and during the path, we also check if any announcement or hiders appear in seeker’s vision. If yes, we will go straight forward if it is hider, or we will check the surrounding cell if it is announcement by using **A** **Star** algorithm.

* The formula of calculating the most value cell will be base on idea that we will find the clearest cell that have a big number of walls around, to do that we will need to find: number of walls arround that cell and diagonal distance from the seeker position to that cell, but for big map size, we have to choose an efficience coefficient for number of wall around. After many test cases, we have decided to multiply the number of cell to maximum value of row and column and divided by 16 (because we have maximum 8 wall around hider) then minus for the diagonal distance (so that a cell which is far from seeker will have lower value).

def calcCellValue(cur\_row, cur\_col, goal\_row, goal\_col, board, ROW, COL):

return findNumberOfWallAround(goal\_row, goal\_col, board, ROW, COL) \* max(ROW, COL) // 16 - findDiagonalDistance(cur\_row,cur\_col, goal\_row, goal\_col)

* For **A\*** algorithms, we will implement it like the a normal **A\*** algorithm, every new state is a new Map with **weight** is increased by 1 if that new state is moving to an unvisited cell, and increase weight by 2 if that new state is moving to a visited cell. For **heuristic value**, we will set it as **diagonal distance**, which is a assistency heuristic. We also have implemented class **PriorityQueueElement** for generating lightest state in frontier. This A\* algorithm return a path to current goal, and for every state in a path, we will check if hider is in seeker vision, if yes we will go straight to it by running A\* for one more time.

When we reach our destination – **the** **most** **value** **cell**, we use **local** **search** to decide where to go next. Our local search will choose the direction which we can explore the most cell that seeker has’t visited or seen. For example:

A screenshot of a game

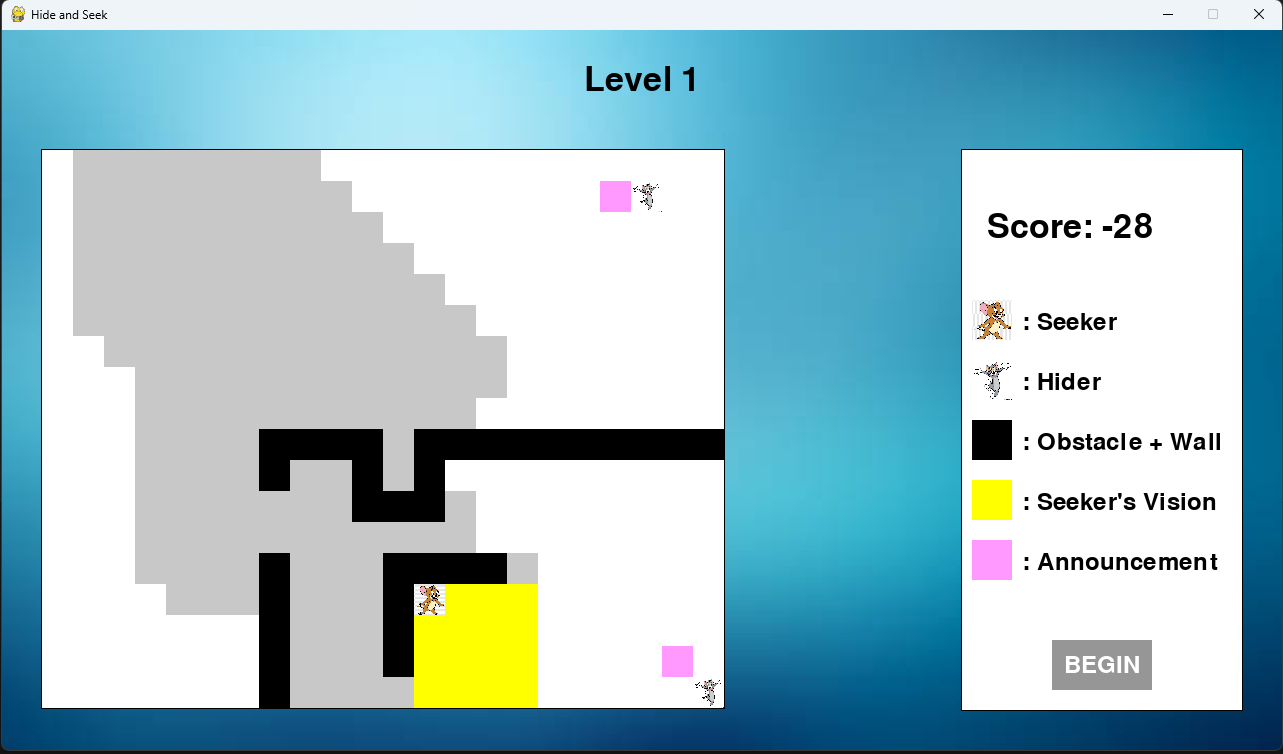
Description automatically generated

In this case, we will calculate the most value cell, which is at the position of the seeker in the picture. Then we use local search to explore the map. However, the only way the seeker can move is go up, and which mean the new cells that seeker can explore don’t increase. We call that situation is the flat or the local-max. To solve this problem, we don’t use local, we will repeat the first step, which calculate the new most value cell the go to that position then use local search, if it still be stuck, it will repeat from the first step again. In the current example, it will find the next most value cell.

A screenshot of a game

Description automatically generated

After find the **2nd** **most** **value** **cell**, it is still stuck, so it continues to find the next most value cell.



In the **3rd** time, it isn’t stuck anymore and use **local** **search** to explore the map.

A screenshot of a game

Description automatically generated

Then, during his exploration with **local** **search**, it find the hider so it will go straight to the hider and catch it. If you want to watch his path or which direction does he choose, please run our program for more details.

Lets move through the **announcement** and how it is created. We create **announcement** at the beginning of the game, and the **announcement** will last for 5 steps, after 5 steps, it will generate a new **announcement**.

## Level 3:

For general, level 3 uses almost everything in level 1 – 2, we will have the same **getVision()**. **moveSeeker()** and **A\*()**. Because announcement make the board hard to handling, we dicided to not creating announcement. For solving this level: First, we will use A\* recursively to visit new state to check for hider, and hider will move randomly too, with a new function **moveHider()**:

* **moveHider():** a function to create a random move for hider by using function random library, it will return a position ([row,col]) represent new position of hider and we will assign it to current Map.

Wheneverseeker see hider, we will have another function to move seeker is **moveSeeker2():**

* **moveSeeker2():** This function will generate new moves and find a move that have **least Mahhatan distance** to the last position of hider that seeker had found and return it. By this way we will always have the best choice for a new move to get closer to the hider