HO CHI MINH NATIONAL UNIVERSITY

UNIVERSITY OF SCIENCE

---o0o---

Introduction to Artificial Intelligent

<u>Project 01</u>

HIDE AND SEEK

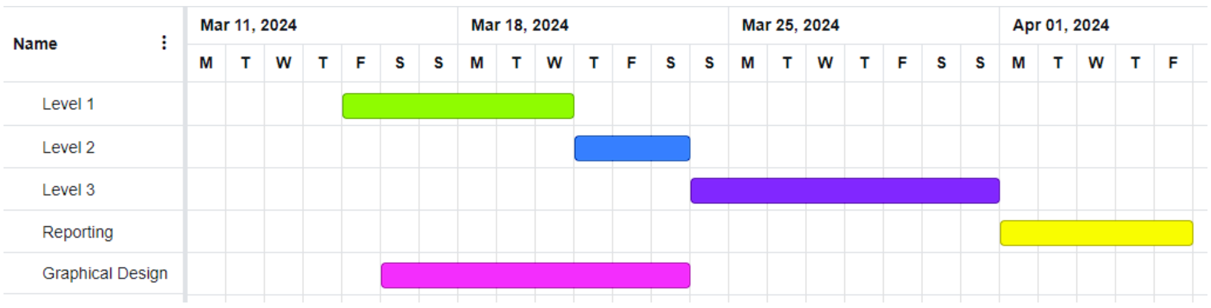
Instructors:	Nguyễn Ngọc Thảo	Hồ Thị Thanh Tuyến
	Lê Ngọc Thành	Nguyễn Trần Duy Minh

Members:	Trần Anh Minh	22127275
	Đoàn Đặng Phương Nam	22127280
	Bùi Nguyễn Lan Vy	22127465
	Diệp Gia Huy	22127475

HO CHI MINH CITY, APRIL, 2024

CONTENTS

I. Assignment Planner



II. Completion

Criteria	Completion
Finish Level 1	100%
Finish Level 2	100%
Finish Level 3	100%
Finish Level 4	Did not implemented
Graphical demonstration	100%
Maps generations	100%

III. Environment Requirement

Python version: 3.10+ with Pygame graphical module.
Consider installing Pygame using *pip install pygame* if it is not available.

Usage: Run the application by executing *python main.py* in the console/terminal.
Notice: The command may vary across platforms, the above command it tested on a Windows operating system.

IV. Maps Design

To some extent, generating maps for testing the seeker has lots of things in common with letting the player find the way to solve a maze. Creating multiple paths around and placing obstacles at the correct position would require the seeker to run around the map several times.

When designing a map, avoid creating a path that allow the player to move diagonally across the walls. Placing the hiders in corners, making them out of sight as for the seeker will ensure the seeker to run around and will need to run inside the narrow aisle to find them.

V. Problem

a. Preparation

i. Class Map

The **Map** class plays a crucial role in defining the layout and content of the Hide and Seek game environment. This map was defined in the file *Map.py*.

- Upon initialization, the class reads the provided matrix, extracting essential details such as the number of rows and columns to establish the map's dimensions.
- Additionally, it identifies the positions of both the seeker and all hiders within the map, storing this information for subsequent use and update during gameplay.

ii. A algorithm*

Being implemented in *A_Star.py*, A* is an irreplaceable algorithm for finding the shortest path from a starting position to a goal position, which helps the seeker move to a certain objective with the minimum cost after having identified it before and becomes one of the criteria for evaluating the effectiveness of the heuristic function in our product.

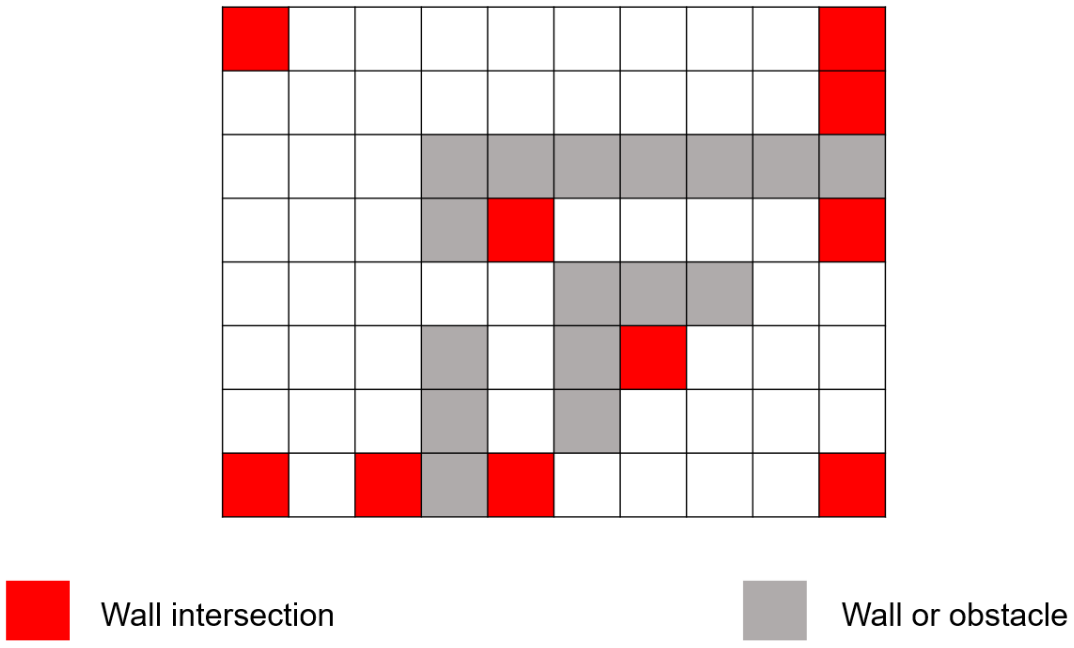
Before the A* algorithm is programmed, the **Node** class is of great significance in encapsulating the properties and behaviors of nodes within the state space of the search strategy. Besides indispensable attributes such as the state, the parent, and the goal state, each node also receives a Boolean argument ***visited***, which is marked as 1 if the node has visited before in the map and 0 otherwise.

- This argument is supplemented in our class **Node** to support our strategy in level 1 and 2 of the game, even at higher levels. In the evaluation function of each node, we will **add 100** if the node is visited. The main purpose of this action is to create a moderate penalty for agents in the game, helping them to avoid moving to visited cells and finding hiders in smaller number of steps.
- This is specifically effective in the first 2 levels, when hiders have no rights to move on the map, so if cells in the seeker's observative ability include no hiders, we can mark them as visited and avoid moving to them in the next steps. But in higher levels, when hiders can move quite optimally, the above idea cannot be maintained successively during the game, so to create a generalization for each node, we will assign 0 to the argument ***visited*** by default.

iii. Wall Intersection

From now on, the definition "wall intersection" is repeated many times in our report, so for the purpose of giving a clear view of this terminology, we will reserve a part for it.

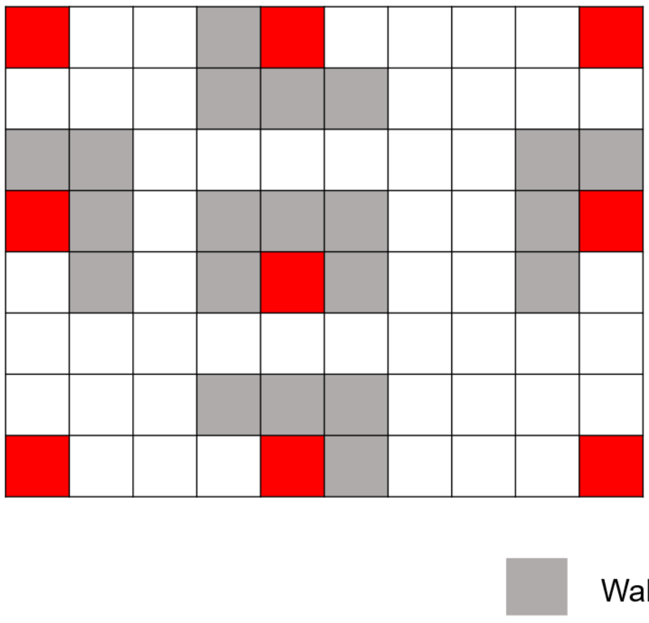
In general, in a 2x2 matrix, if 3 of 4 cells are walls or obstacles, the remaining cell will be a wall intersection. Moreover, if a cell is located on the edge of the game map, it can be considered a wall intersection; it is one of four corners of the map, or its location is aside from a wall or an obstacle. Below is a demonstration of wall intersections on a particular map.



We named this definition "wall intersection" instead of "wall corner" because we consider a corner to be a special wall intersection:

1. It can be surrounded by three walls or obstacles.
2. It can be one of four corners of the map.
3. If it is in the edge of the map, it can be surrounded by two walls or obstacles.

Below is a demonstration of corners on a specific map:



iv. Class Level

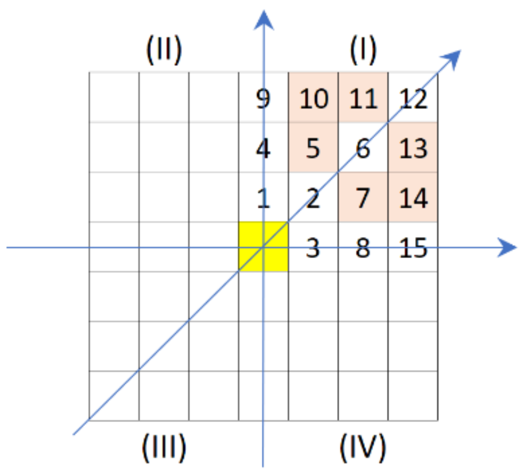
To prepare common attributes and methods that can be utilized in all levels of the Hide and Seek game, we create a versatile class called **Level**, and then in each level, we will simply create the corresponding class which inherits this class besides supplementing essential things for each specific level.

Level class efficiently initializes a general problem space with the map and score of the game, the number of steps taken by the seeker and all of the hiders, as well as the current turn-taker, who can be the seeker or hider. In addition, this class also defines the following methods:

broadcastAnnouncement: Receiving the hider's position, which has the demand to broadcast an announcement on the map as the only additional input, this method returns a random position for the hider within radius 3, including potential obstacles or walls.

getObservableCells: This method is used to obtain observable cells within radius 3 that have the current position of the seeker. The flow of this function is split into three parts:

- Firstly, we create a Boolean matrix with a maximum size of 7x7 around the seeker position, not including cells whose index is outside the range of the game map. We initially assign false to the cells that are walls or obstacles and true to the other cells. Through instantiating this matrix, we can get the exact position of the seeker in the matrix, which helps us identify the observable cells.
- After that, by sequentially considering cells in the radius of 1 and 2, following the requirement of the teachers, we consider all cases that can occur, making a cell unobservable, and assign false to all of the unobservable cells in the matrix.
- Finally, from the Boolean matrix, we can get the observable cells of the seeker in the original map.



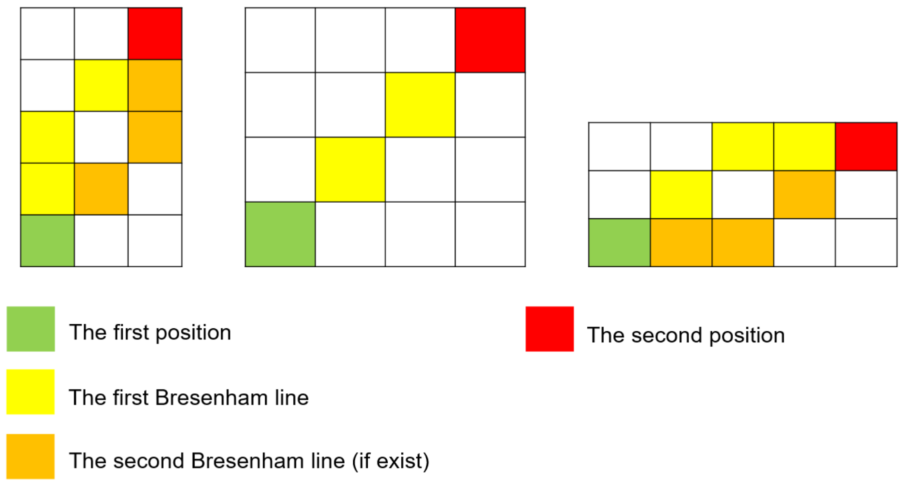
Principle in obtaining observable cells of the seeker

(copied from teacher's guideline)

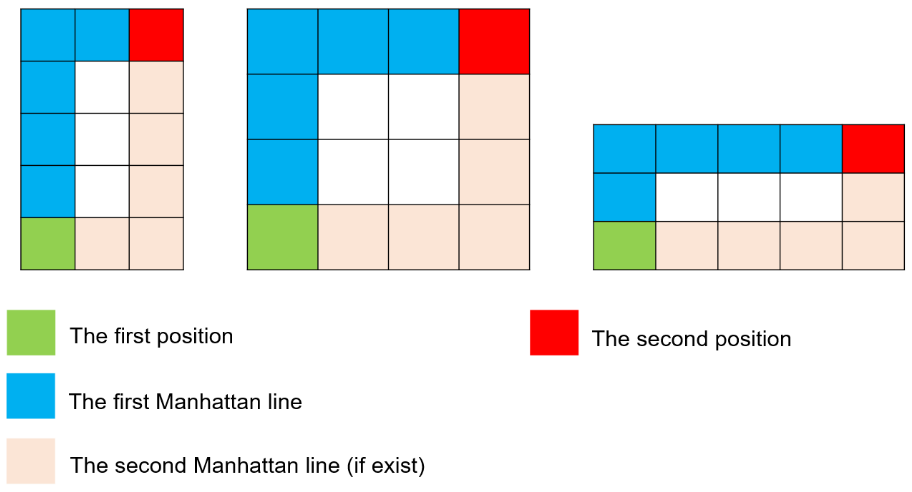
getShortestPath: Being such an application of the A* algorithm after the algorithm's implementation, this method receives the starting and goal positions as indispensable arguments for the A_Star function, besides the parameter ***visitedMatrix***, which can be used depending on the demands of different situations. The purpose of this argument was presented in this part, and by default, the argument ***visited*** is assigned to **None**.

countNumWallsBetweenTwoPositions: Obtaining two arbitrary positions as inputs, the return value of this method is the minimum number of walls or obstacles in separated paths between two positions. Particularly in this function, we consider two types of lines: **Bresenham lines and Manhattan lines**.

Below will be images for the demonstration of two kinds of lines. We will suppose that the first position is under and on the left of the second position. The similarity is true with the three remaining correlations of the two positions



Bresenham lines



Manhattan lines

checkCorner: Receiving a position in the map as the only input, this function returns True if the position is a corner; otherwise, it returns False.

The two above functions are of notable importance in creating a heuristic function for evaluating the nearest wall intersection from a certain position in the map, which is one of the main approaches in all strategies of levels in the Hide and Seek game.

v. Supporting functions

During the process of completing each level of the game, any supporting functions arising will be implemented in the file util.py, besides constant values. Some considerable functions that can be mentioned are:

getValidNeighbors: This function will return all the valid neighbors of a cell in a map, which can play the same role as generating successors in implementing the A* algorithm.

getTrendMoveDirection: Based on the angle between the vector from the starting position to the goal position, this function identifies the trend of moving, which is one of the 8 directions with the least angles with the above vector. The returning value of this function will be one of 8 strings: "UP", "UPRIGHT", "RIGHT", "DOWNRIGHT", "DOWN", "DOWNLEFT", "LEFT", "UPLEFT".

setOrderOfNeighbor: Based on the trend of movement that was identified in the function ***getTrendMoveDirection***, the function will receive a list of neighbors and change it in clockwise order from the moving trend. This helps save the cost of finding the best moving for both hiders and seeker in Level 3.

vi. Convention

Before moving on to our strategies for levels in the Hide and Seek game, we want to present a little bit about our convention for the announcement of each hider.

Each announcement will be broadcast every 8 steps from the start time of the game and exist for 2 steps before disappearing.

b. Level 1

Our strategy for level 1 is to sequentially find the nearest wall intersection from the current position on the map. When you reach a certain wall intersection:

- If there is no hider around this wall position, we will move to the next nearest wall intersection.
- If there is a hider, touch that hider.

Through the process of moving among other wall intersections, we keep a matrix for checking visited cells that are identified as not including hiders, and we hope that we can find the only hider through the process.

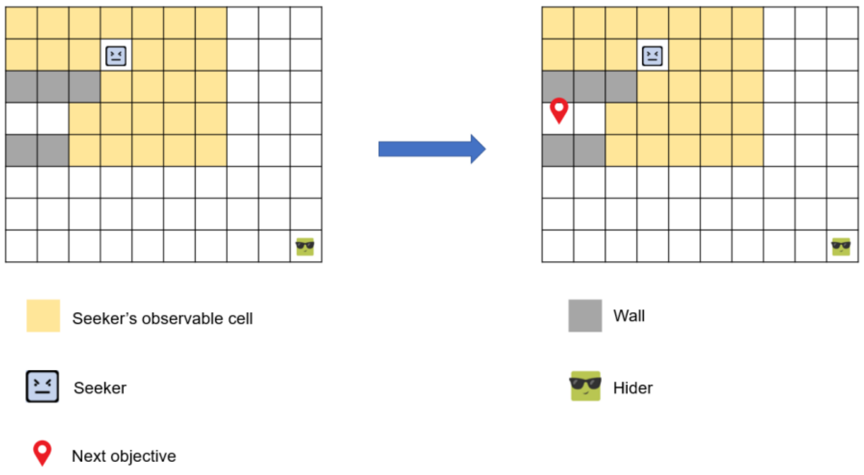


Fig: There is not hider around → move to the next unvisited nearest wall intersection

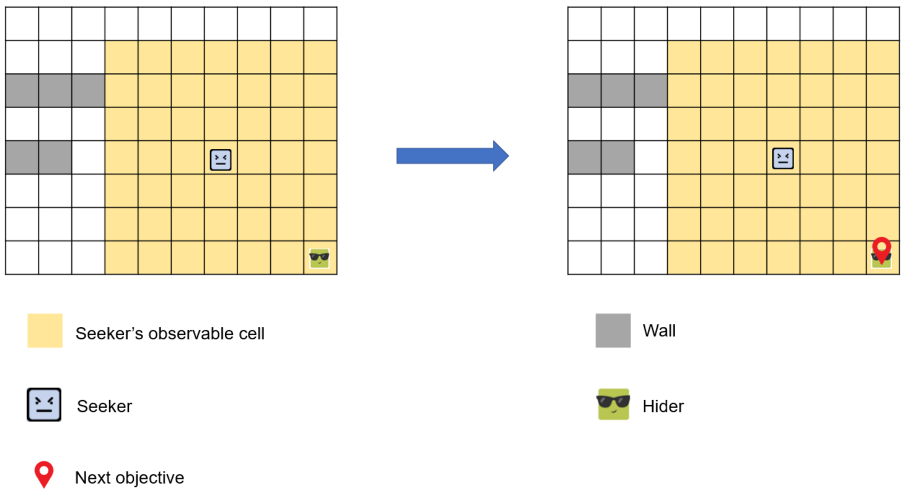


Fig: If there is hider around → Conduct to touch that hider

On this strategy, we optimize the process of finding the nearest wall intersection by building a heuristic function to estimate the distance from the current position to the wall intersection, based on two elements:

- The number of walls between the current position and the wall intersection (on Bresenham lines and Manhattan lines)
- Whether the wall intersection is a corner or not.

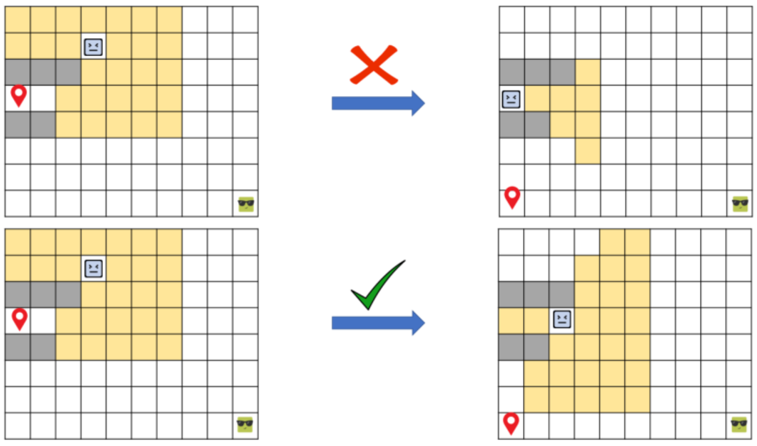
Our heuristic function is: h = d - 0.9 * isCorner with **d** is the distance from the start position to the wall intersection and **isCorner** is 1 if the wall intersection is a corner, otherwise, it is 0.

0.9 can be replaced by another value, which is less than 1, because:

- We can prioritize the wall intersections which are corners if the value d of two wall intersections is the same.
- 0.9 or any value less than 1 can reduce the ambiguity of the equal heuristic values.

If two wall intersections have the same heuristic value, we will choose the wall intersection which has the shorter path, calculated by the method **getShortestPath.**

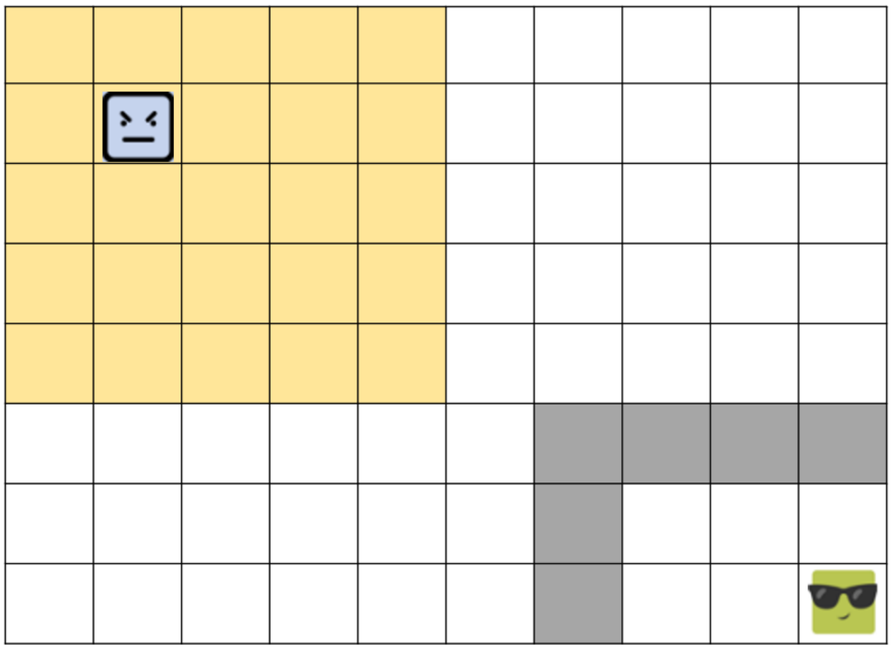
Besides, to optimize the score of game, instead of moving to the cell that is identified the nearest wall intersection, if that cell is in observable cells of the seeker and it does not include the hider, we will move to the next nearest wall intersection.



On the road to move among wall intersections, if the seeker can observe the hider or the announcement, it will conduct to touch the hider.

If an intersection is identified but there are no paths to go to that intersection, we will ignore that intersection. After visiting all wall intersections, if no hiders are found, we will check unvisited cells and move sequentially to each cell in the order from nearest to farthest.

Additionally, if the seeker realizes that it cannot find the hider (there are no ways to go to the remaining hiders), it will give up.



Seeker gives up

c. Level 2

Our strategy for level 2 is nearly the same as for level 1, only different in three points:

- When the seeker realizes that it cannot find a certain hider, instead of giving up immediately, the seeker will ignore that hider and start to move on to the next objective. The seeker will give up when the number of ignored hiders is equal to the number of hiders currently on the map.
- When many hiders are identified simultaneously, the seeker will choose the first hider added to the list of identified hiders and conduct a touch on that hider. After being touched, the hider will be discarded from the list of hiders on the map as well as in the seeker's perception. Then the seeker will continue sequentially with the remaining hiders.
- With announcements, in level 2, we create a dictionary mapping each position of announcement with the position of corresponding hiders (because two hiders can broadcast announcements at the same position). In existing time, if the seeker observes one or many announcements, the seeker can "look up" this dictionary to identify corresponding positions of hiders and add them to the list of identified hiders.

d. Level 3

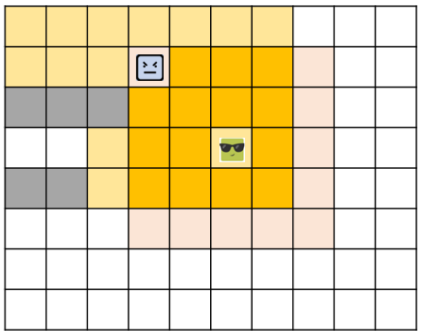
Basically, our strategy for level 3 is a mixture of strategies for level 1 and 2, when the seeker initially identifies the nearest wall intersection to find hiders, besides maintaining a list of ignored hiders to give up reasonably. Despite many identical points, the uniqueness of our strategy for level 3 compared to two previous levels is denoted by the following ideas:

- The seeker still maintains a matrix of visited cells, but it does not consider this attribute in finding the shortest path to the nearest wall intersection or hiders and announcements. The visited matrix can just be used to mark unvisited wall intersections or when the seeker wants to move to a certain cell if all the wall intersections are visited. The reason for this decision is because of the movement of hiders, which share unknown information with the seeker, so the seeker will not be adventurous in not revisiting visited cells unless they have enough basis.
- Because at this level, hiders can move, a class called Hider will be created to store the properties and methods of each hider, independently of the state of the seeker. Each hider will have its own method to identify observable cells as well as identify whether there is a seeker in its observable cells. Besides, hiders will be distinguished through an ID number for each hider.
- When the seeker and one of the hiders meet each other, both of them will have demands to move to the next cell, which is the most difficult for the opponent. So our strategy is to define a method called **getBestMoveWhenHiderMeetSeeker**. This function helps the seeker as well as the hiders make the best moves when they meet each other. Based on the idea of "Nash Theory" in Game Theory, we will build a table that stores tuples of values.
    - If the player is the seeker, then each tuple of value will be *(x,y)*, with:
        - *x:* The length of the shortest path from seeker to hider if the seeker move to a certain cell around it
        - *y:* The length of the shortest path from hider to seeker if the hider move to a certain cell around it after the seeker has taken their turn before (i.e. the movement corresponding the value x)
    - If the player is the hider, the idea of building the table is the same with the above explanation, we just swap the role of two players.

We will choose the best move such that:

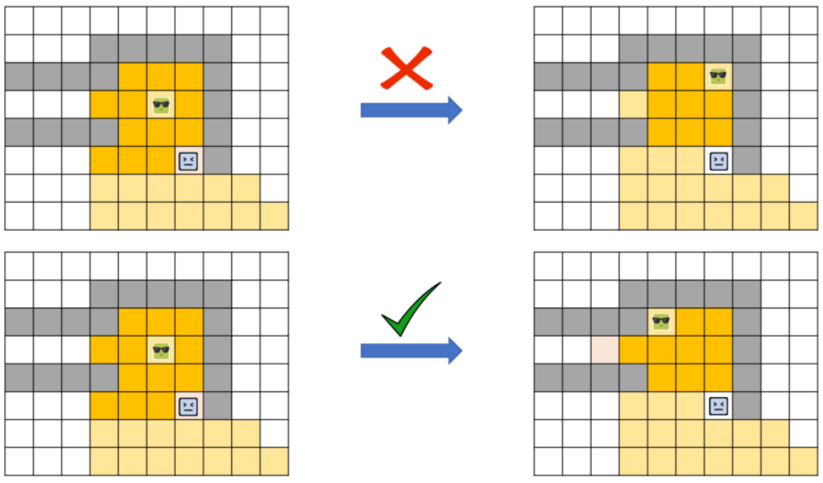
- Firstly, in each move of the first taker (who can be a seeker or a hider), we get all of the tuples (x,y), with y being the best option for the second taker.
- After taking all the tuples satisfying the above statement, we will choose tuples with x as the best option for the first taker.
- If there are many tuples satisfying that x is the best option, choose tuples with y as the worst option for the second taker.

- Until now, if there are many tuples satisfying the above statement, we have considered two cases:
    - If the first taker is the seeker, we will choose the cell where the number of cells the hider can move to not be caught in the next move is the smallest.



Seeker's move direction	Number of hider's escaping cells next move
UP	8
UP - RIGHT	8
RIGHT	6
DOWN - RIGHT	5
DOWN	6
LEFT	8
UP - LEFT	8

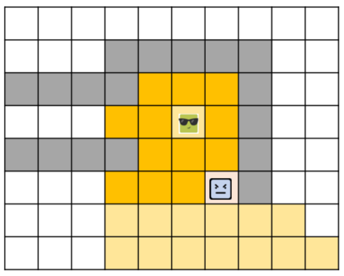
- If the first taker is the hider, we will sequentially consider the following conditions, prioritized from the highest to the lowest:
    - If the cell is not a wall intersection or is a wall intersection and we still move diagonally if moving to that intersection, choose that cell.

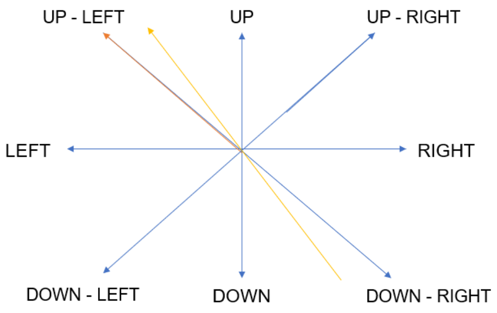


- If the cell can allow the hider to have the largest number of escape cells (we suppose that the hider thinks that the seeker will minimize the number of cells the hider can move to not be caught in the next move), choose that cell.
- If the cell can allow the hider to observe the largest number of cells, excluding wall intersections, choose that cell.

Besides, in valid neighbors of the hider, when the taker is a hider, we will discard the neighbors, which are the positions of other hiders, to avoid collapsing.

We also set the order of the neighbors of the current hider position when the hider takes a turn based on the trend of the movement from the seeker to the hider, which can help the hider consider cells forward instead of towards the seeker.





-> The trendy of move direction is UP - LEFT

-> (UP – LEFT, UP, UP – RIGHT, RIGHT, DOWN – RIGHT, DOWN, DOWN – LEFT , LEFT)

- With hiders, we also consider the following cases:
  - If the hider is not being chased by the seeker, the hider will make the best move to maximize the number of observable cells. This helps the hider avoid standing at corners or wall intersections, which can be easily caught by the seeker.
  - If the hider is being chased by the seeker, the hider will make the best move to avoid being caught by the seeker based on the method **getBestMoveWhenHiderMeetSeeker**.

Additionally, if the hider is being chased by the seeker but cannot observe the seeker, we also define an attribute **isBeingChased** which will help the hider realize that it is still being chased by the seeker and choose the best move for the next step.

e. Level 4

Though we did not implement level 4, we have some ideas about this.

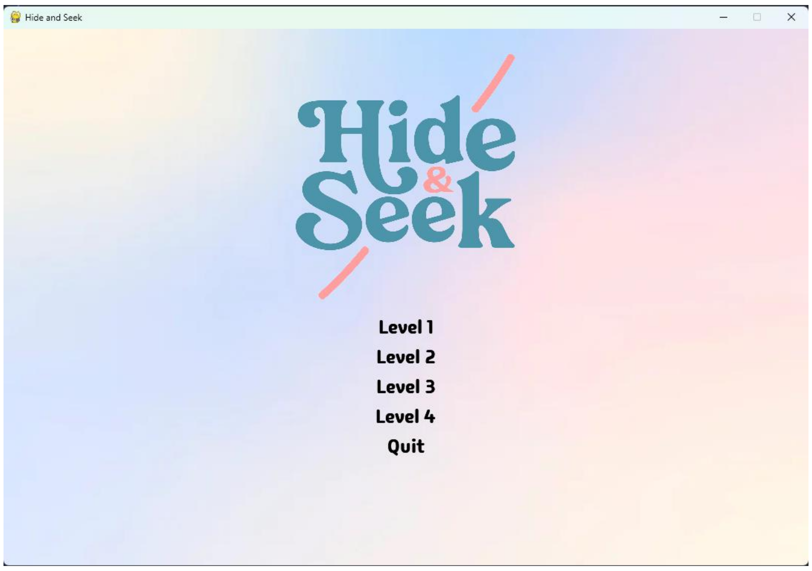
One opinion is that, letting the hiders identify the corners, they should move the obstacles to fill up the entrance and hide inside. Therefore, it blocks the seeker from finding itself.

Another idea is to allow hiders to move the obstacles on their way, running away from the seeker if being chased, and block the path to have the seeker find another path to reach it.

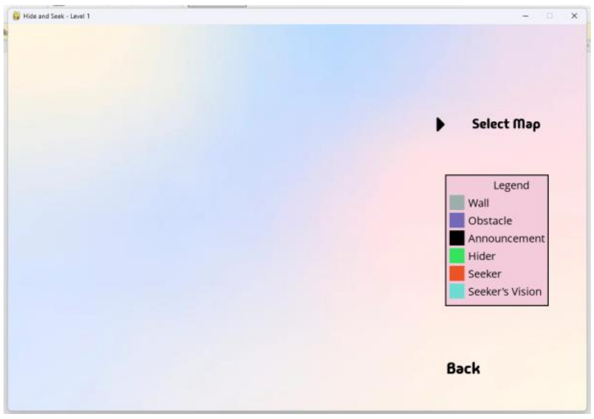
However, blocking the path in the first idea may not seem like a valid choice for a non-reinforcement approach; the hiders need to study the map well enough to figure out where the dead end is located.

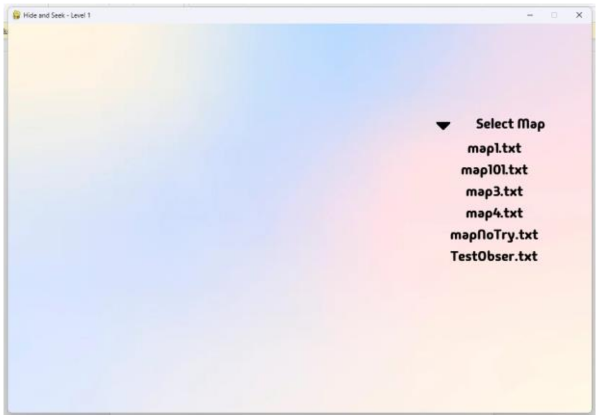
VI. User Interface and Game Play

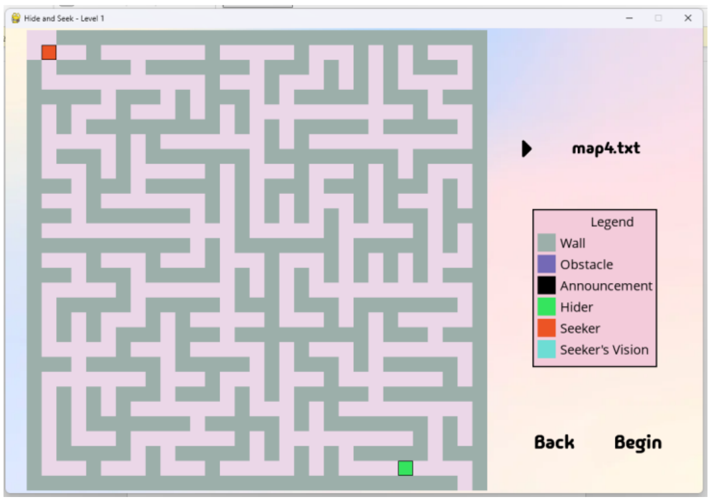
When running the program, a screen will pop up to allow the user to choose the desired level to run.



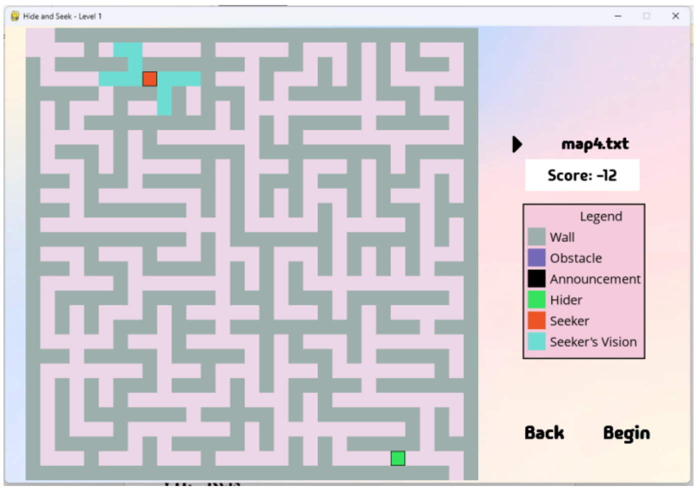
After choosing a level to run, the screen changes, and a button is shown to let the user pick a map to run. When clicked, all available maps are provided for the user to choose and it will appear accordingly.



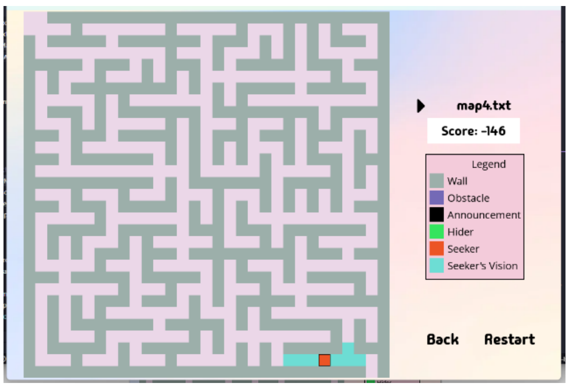


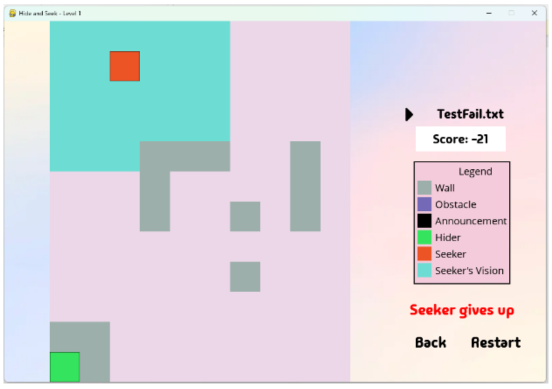


Only then will a begin button pop up, allowing the user to run the game.



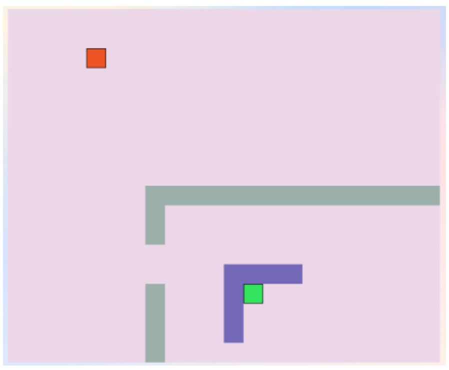
There can be two results: the seeker will find all hiders, or the seeker will give up (hiders can make the seeker chased endlessly or there is no path to reach them). And the begin button will be changed to "Restart", letting the user load the map again for another run.





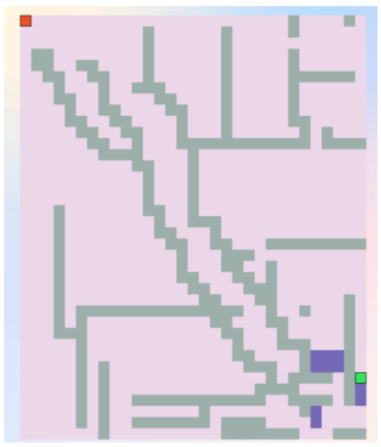
VII. Testing and Commenting

a. Level 1: Single – hider



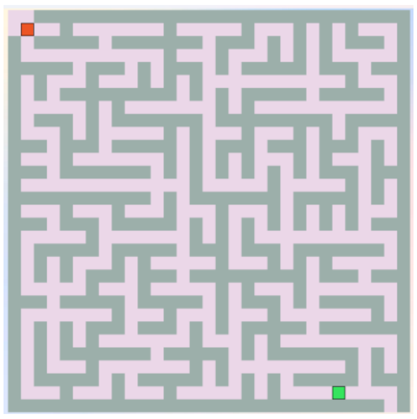
Size: 18x22

Score: -24



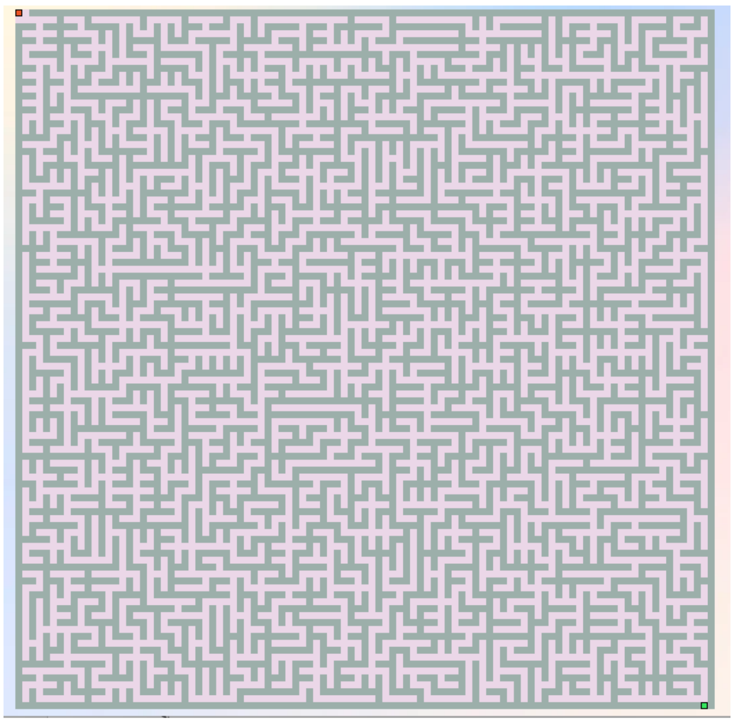
Size: 38x31

Score: -268



Size: 31x31

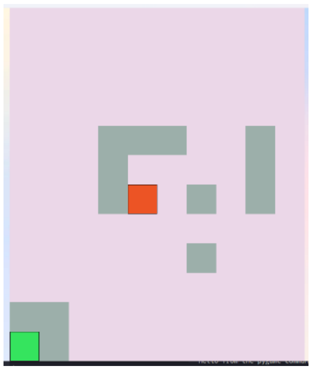
Score: -146



Size: 101x101

Score: -3038

Our implementation requires the seeker to visit all squares in the map before giving up; therefore, in maps that are very large, it takes a very long time to reach the location of the hider.

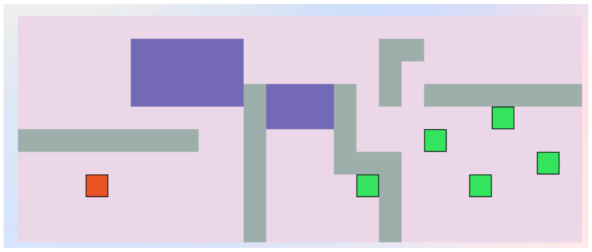


Cases when there is no path to reach the hider, the seeker will run around the maps to search for the aisle and eventually gives up after visited all possible squares.

The score of the map: -21 and seeker gave up.

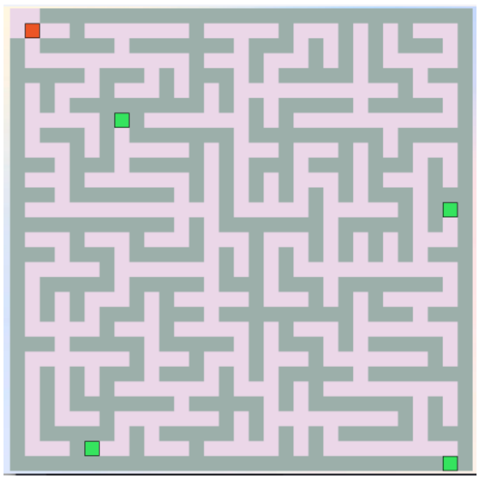
We have added in a 199x199 map, but due to the complexity of running and displaying it (too small), we will not list it in this evaluation.

b. Level 2: Multi – hiders



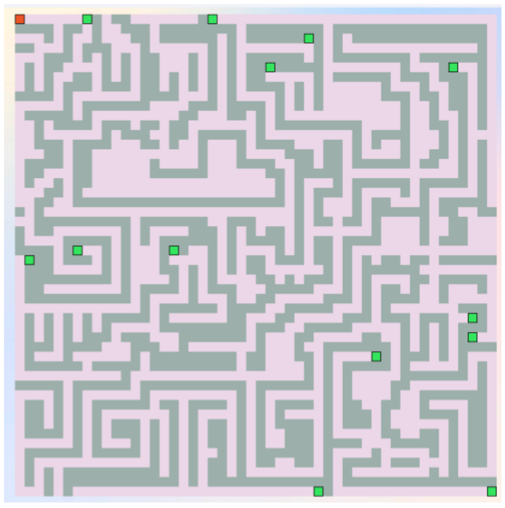
The seeker gave up after find the 4 hiders on the rightmost side, and find no path to reach the last one.

The final score: 41



Size: 31x31

Score: -228



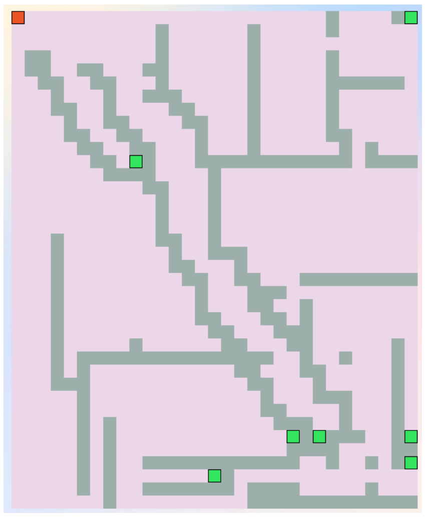
Size: 50x50

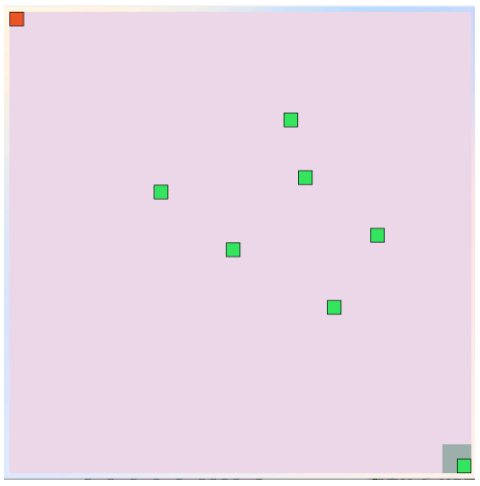
Score: -1220

As the seeker also needs to visit all the squares, all the hiders will eventually be found.

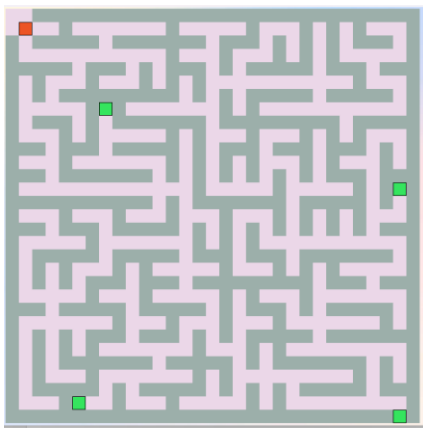
Though the walls form a maze, the seeker movements may miss some squares in the corners, resulting in much more steps being required to visit those squares of the leftovers.

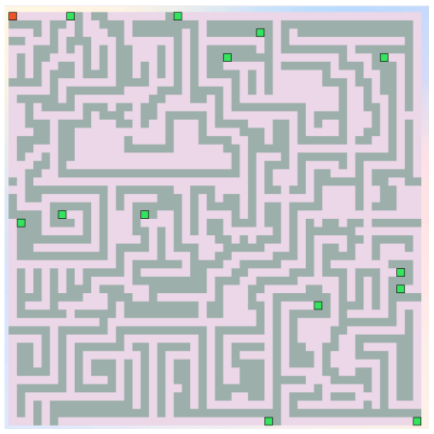
c. *Level 3: Multi – movable – hiders*


Score: -238


Score: -163

By allowing hiders to move, their first steps will make them move to the squares that have the most observable cells around them. Seeing the seeker, they will move in the opposite direction in order to run away from him. Testing the hide and seek game on a map with no wall shows all possibilities for the seeker's chasing strategy and the hider's method of moving away.


Score: -315


Score: -1176

These maps are exactly the same as the ones in level 2, but now that the hiders can run, the game is much more fun seeing them chase around the map, resulting in worse scores.

VIII. Demo Video and Reference

a. Demonstration Video

https://drive.google.com/file/d/1FfvFUIVgkkNwVr5YCtRrc7gdDps8ibAp/view?usp=sharing

b. References

https://www.pygame.org/docs/ref/pygame.html