HO CHI MINH NATIONAL UNIVERSITY

UNIVERSITY OF SCIENCE

---o0o---

**Introduction to Artificial Intelligent**

**A blue and white logo

Description automatically generated**

**Project 01**

**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**

# Assignment Planner

# 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.

# Idea and Theory

# 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.

# Problem

## Preparation

### Class Map

**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.

### A\* algorithm

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

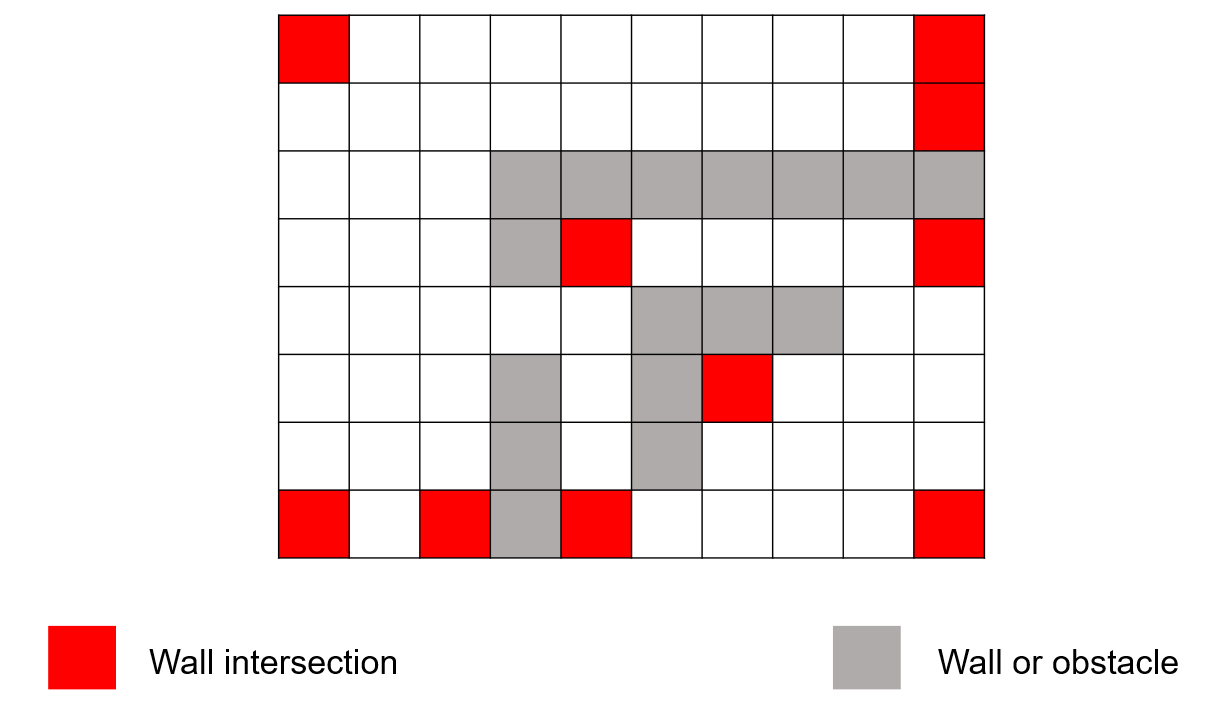
Before A\* algorithm is programmed, **Node** class is of great significance in encapsulating the properties and behaviors of nodes within the state space of the search strategy. Besides indispensible attributes such as the state, the parent as well as the goal state, each node also receives a boolean 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 level 2 of the game, even in 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 creating a moderate penalty for agents in the game, helping them to avoid moving to visited cells and finding hiders in less number of steps.
* This is specifically effective in the first 2 levels, when hider have no rights to move in the map, so if cells in the seeker’s observative ability include no hiders, we can mark them as visited and avoid to move to them in next steps. But in higher levels when hider can move quite optimally, the above idea cannot be maintainable successively during the game, so to create a generalization for each node, we will assign 0 to the argument ***visited*** in default.

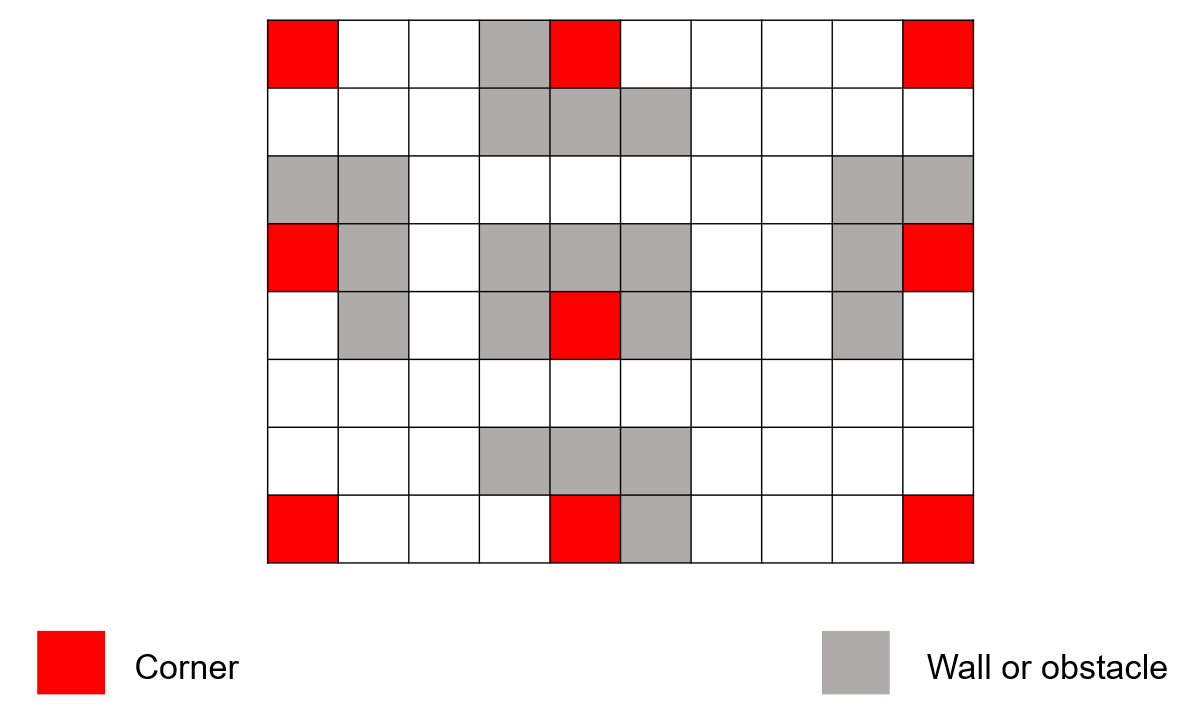
### Wall Intersection

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

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

   
We named this definition “wall intersection” instead of “wall corner” because we consider corner is a special wall intersection:

1. It can be surrounded by 3 walls or obstacles.
2. It can be one of 4 corners of the map.
3. If it is in the edge of the map, it can be surrounded by 2 walls or obstacles.

Below is a demonstration of corners in a specific map:

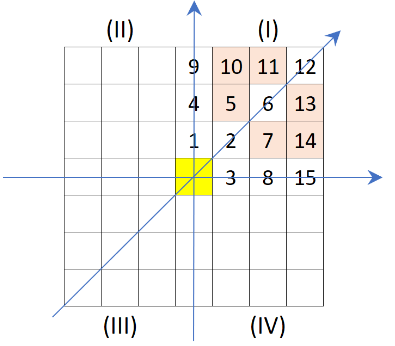
### Class Level

To prepare common attributes and methods which can be utilized in all levels of Hide and Seek game, we create a versatile class called **Level**, 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 game, the number of steps taken by the seeker and all of hiders as well as the current turn taker who can be the seeker or hiders. In addition, this class also defines the following methods:

**broadcastAnnouncement**: Receiving the hider of position which have the demand to broadcast an announcement in 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 having the current position of the seeker. The flow of this function is splitted in 3 parts:

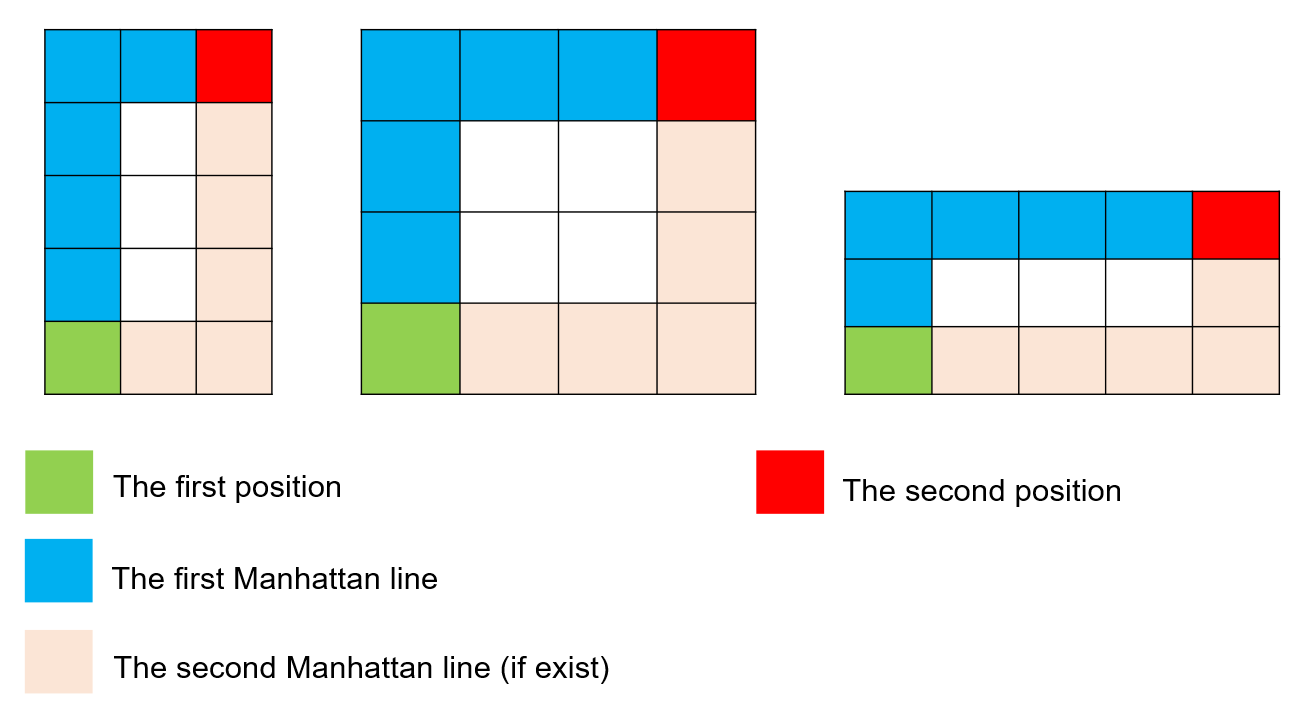
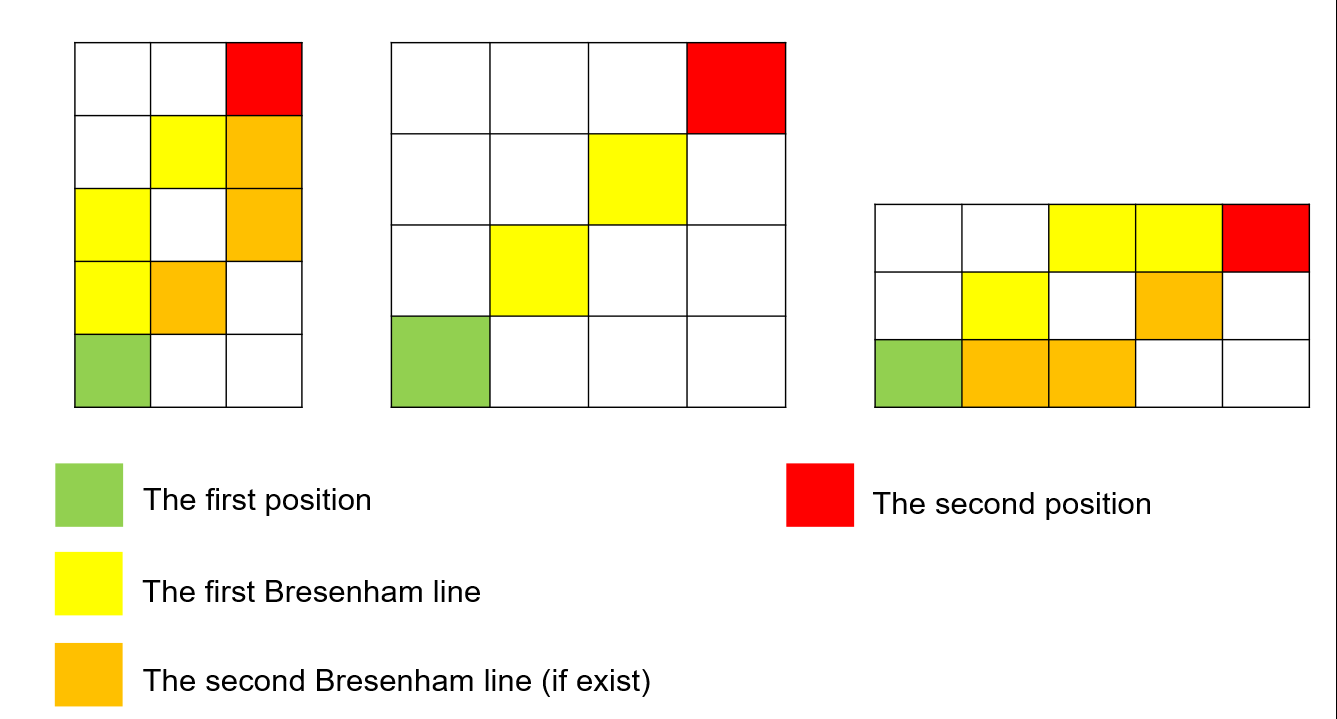
* Firstly, we create a boolean matrix with the maximum size of 7x7 around the seeker position, not including cells with the index being out of the range of the game map. We initially assign False to the cells that are walls or obstacles, and True to the other cells. Through instantiating this matrix, we can get the exact position of the seeker in the matrix, which supports us to identify the observable cells.
* After that, by sequentially considering cells in radius 1 and 2, following the requirement of teacher, we consider all cases can occur making a cell become unobservable and assign False to all of unobservable cells in the matrix.
* Finally, from the boolean matrix, we can get 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 A\* algorithm after having algorithm’s implementation, this method receives the starting and the goal position as indispensable arguments for A\_Star function, besides the parameter ***visitedMatrix*** which can be used depending on the demand of different situations. The purpose of this argument was presented [this part](#_A*_algorithm), and in default, the argument ***visited*** is assigned by **None**.

**countNumWallsBetweenTwoPositions**: Obtaining two arbitrary positions as inputs, the returning 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 line: **Bresenham lines and Manhattan lines**.

Below will be images for the demonstration of two kinds of line, we will suppose that the first position is under and in the left of the second position, the similarity is true with 3 remaining correlations of 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.

Two above functions are of noticably 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.

### Supporting functions

During the process of completing each level of the game, any supporting fucntion arising will be implemented in the file util.py, besides constant values. Some considerable functions which 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 with generating successors in implementing A\* algorithm.

**getTrendMoveDirection**: Based on the angle between the vector from the starting position to the goal position, this function identifies the trendy of moving which is one of 8 directions has 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 trendy of movement which was identified in the function **getTrendMoveDirection**, the function will receive a list of neighbors and change it in the clockwise order from the moving trendy. This helps save the cost of finding the best moving for both hiders and seeker in Level 3.

### Convention

Before moving to our strategies for levels in 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 after each 8 steps from the start time of the game and exist in 2 steps before disappearing.*

## Level 1

Our strategy for level 1 is sequentialy finding the nearest wall intersection from the current position in the map. When reached a certain wall intersection:

* If there is not hider around this wall position, we will move to the next nearest wall intersection.
* If there is hider, conduct to touch that hider.

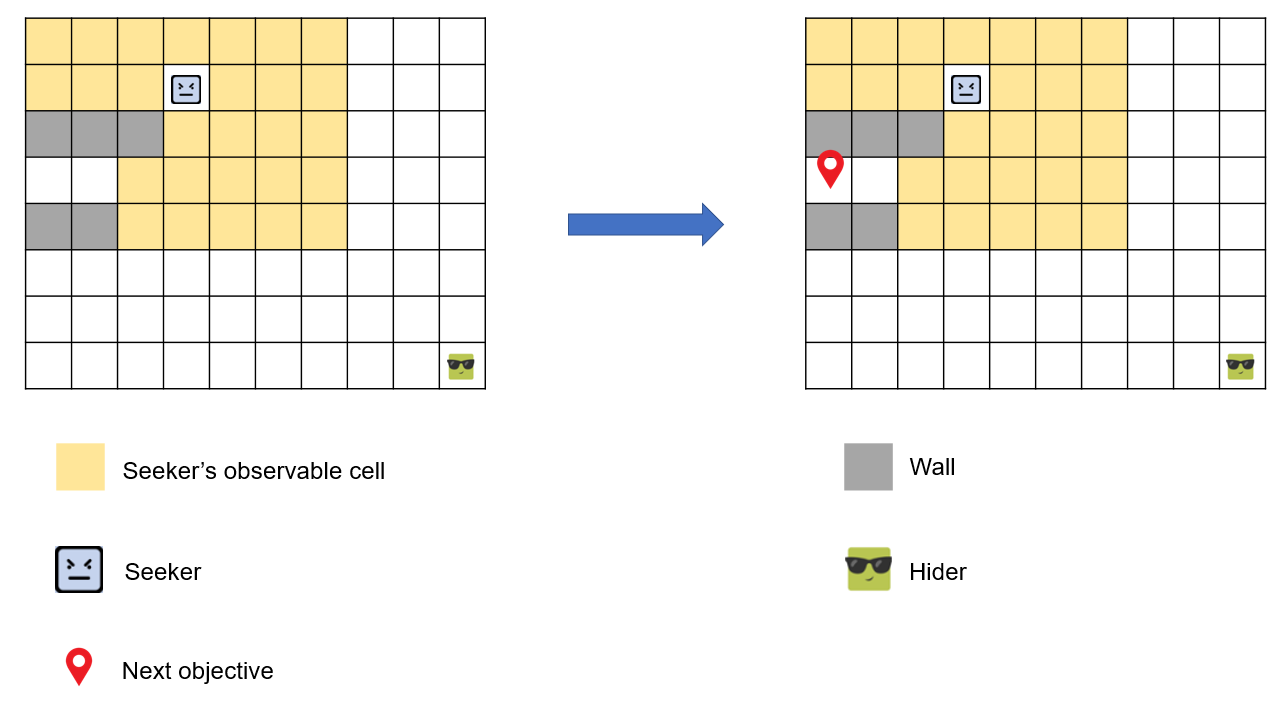
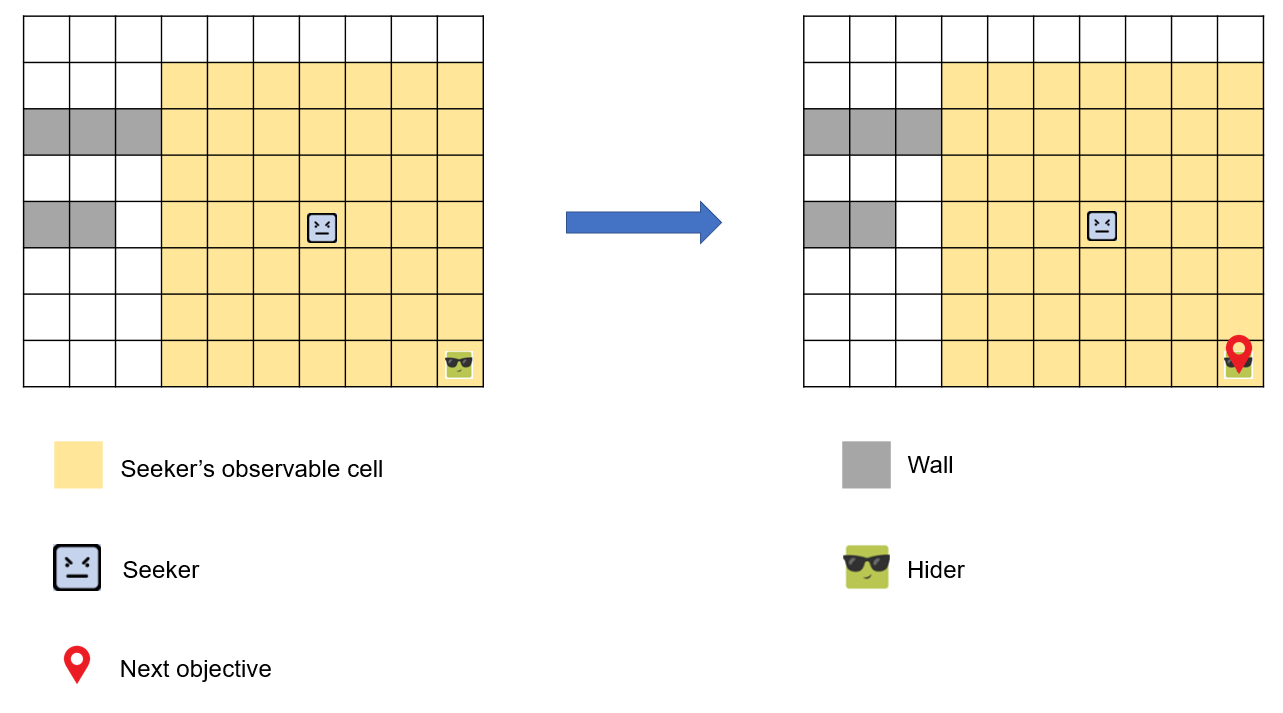
Through the process of moving among other wall intersections, we keep a matrix for checking visited cells which are identified to not include hider, and we hope that we can find the only hider through the process.

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

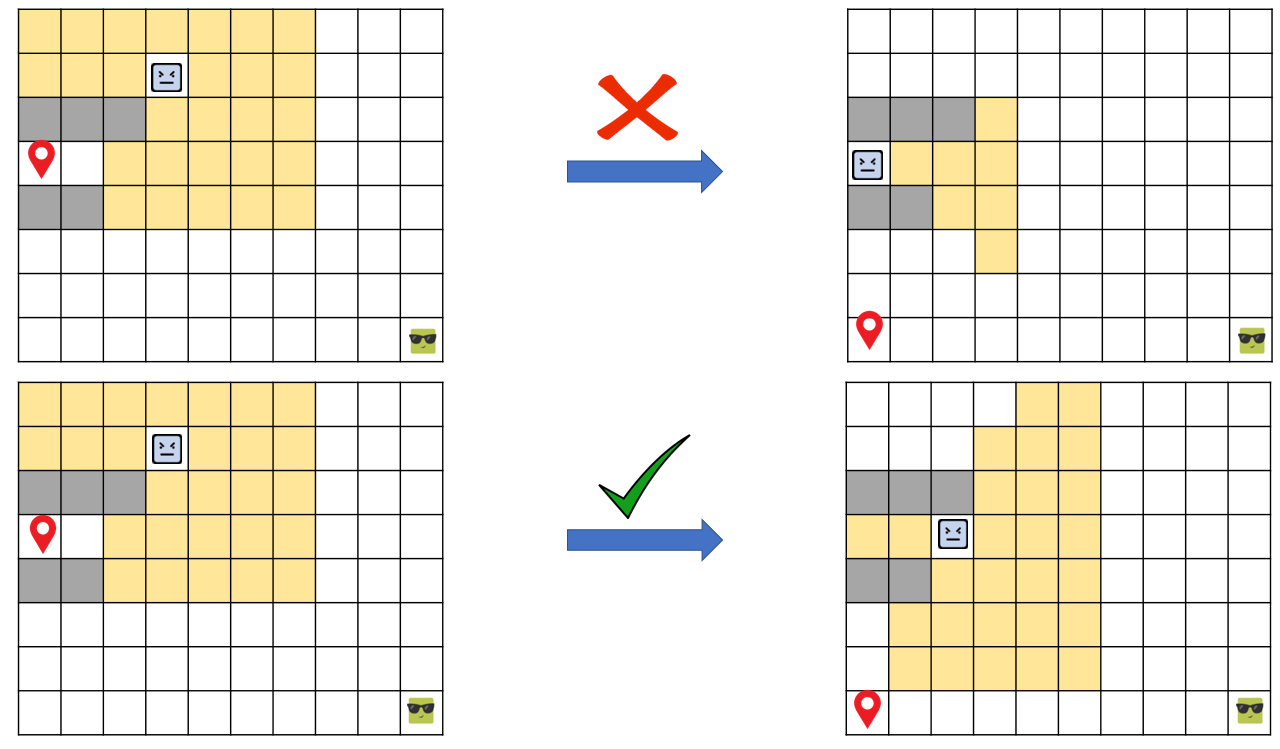
Fig: If there is hider around 🡪 Conduct to touch that hider

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.

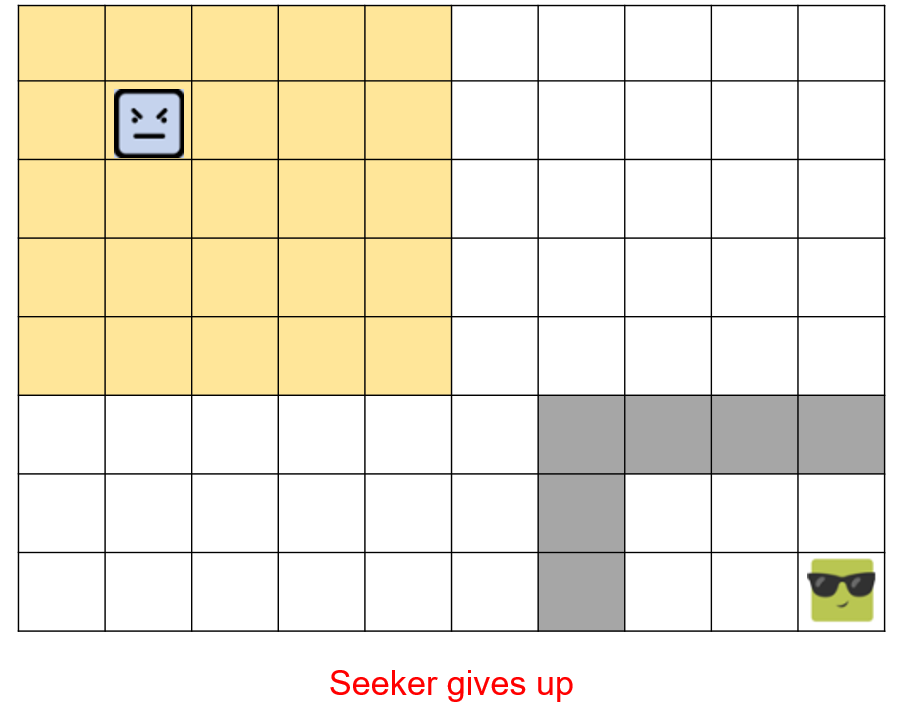
Our heuristic function is: h = d - 0.9 \* isCorner with **d** is the distance from the start position to the wall intersection and **isCorner** is 1 if the wall intersection is a corner, otherwise, it is 0  
0.9 can be replaced by another value, which is less than 1, because:  
+ We can prioritize the 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 which are 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 was identified, but there is no paths to go to that intersection, we will ignore that intersection. After visiting all wall intersections, if there is not any hiders found, we will check unvisited cells and move sequentially to each cell in the order from nearest to furtherest.

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

## Level 2

Our strategy for level 2 is nearly same with 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 to next objectives. The seeker will give up when the number of ignored hiders is equal to the number of hiders currently in the map.
* When identified many hiders simutaneously, the seeker will choose the first hider added to the list of identified hiders and conduct to touch that hider, after touched, reached hider will be discarded from the list of hiders in the map as well as in the seeker’s percept. Then the seeker will continue sequentially will 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 added them to the list of identified hiders.

## Level 3

Basically, out 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 in 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 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 are unknown information with the seeker, so the seeker will not be adventurous in not revisiting visited cells, except having enough basis.
* Because in this level, hiders can move, so a class Hider will be created to store properties and methods of each hider, independently with the state of the seeker. Each hider will have its own method to identify observable cells as well as identify whether there is seeker in its observable cells. Besides, hiders will be distinguished through an ID number for each hider.
* When the seeker and one of hiders meet each other, both of them will have demands to move to the next cell that makes the most difficult for opponent. So our strategy is defining a method called **getBestMoveWhenHiderMeetSeeker**. This function helps seeker as well as hider take the best move when they meet each other. Based on the idea of “Nash Theory” in Game Theory, we will build a table which 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.

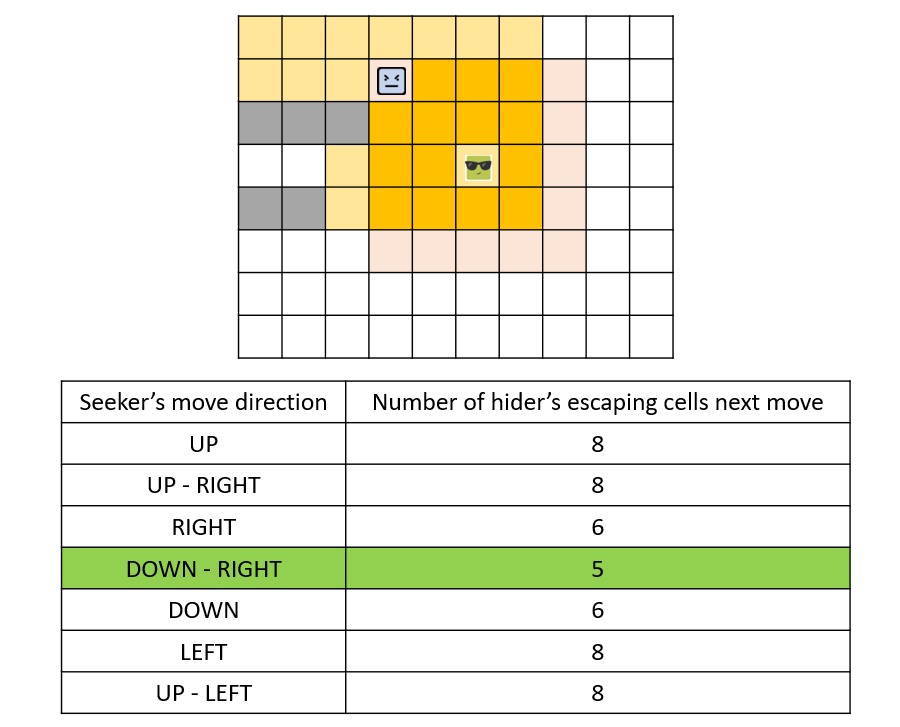
We will choose the best move such that:

+ Firstly, in each move of the first taker (can be seeker or hider), we get all of tuples (x,y) with y is the best option for the second taker

+ After taking all the tuples satisfying the above statement, we will choose tuples with x is the best option for the first taker.

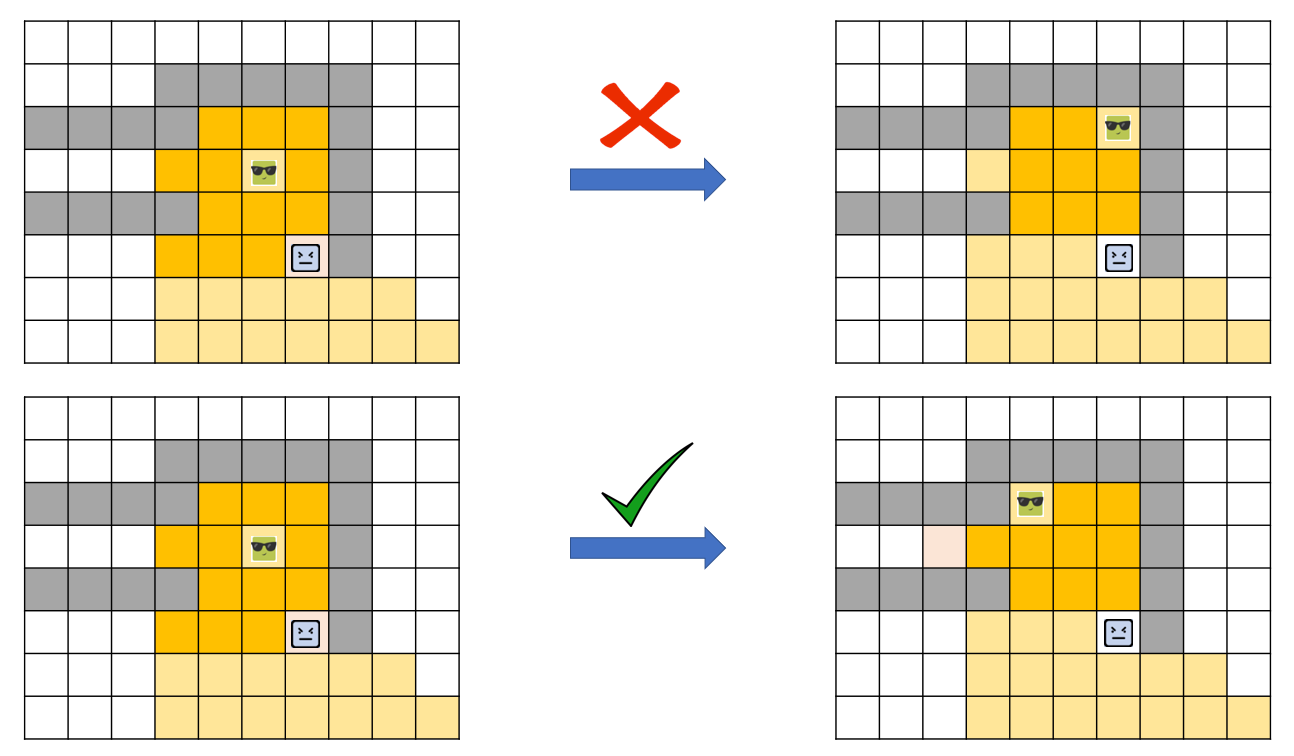
+ If there are many tuples satisfying that x is the best option, choose tuples with y is the worst option for the second taker

+ Until now, if there are many tuples satisfying the above statement, we consider two cases:

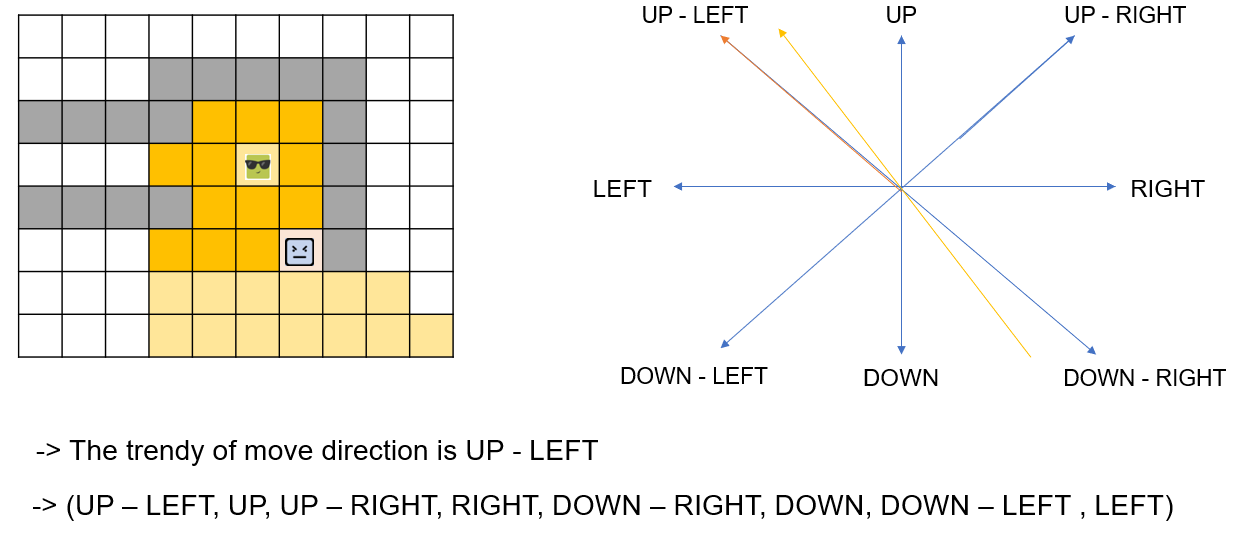
 o If the first taker is the seeker, we will choose the cell that the number of cells the hider can move to not be caught in the next move is the smallest.

o 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 that we still move diagonally if moving to that intersection, choose that cell.



- If the cell can allow 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 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 hider, we will discard the neighbors which are the positions of other hiders to avoid the collapsion.  
We also set the order of the neighbors of the current hider position when hider takes turn based on the trend of the movement from the seeker to the hider,  
which can help hider consider cells forward instead of towards the seeker.

* With hiders, we also consider the following cases:

+ If the hider is not being chased by the seeker, the hider will take the best move to maximize the number of observable cells. This helps hider to 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 take the best move to avoid being caught by the seeker based on the method **getBestMoveWhenHiderMeetSeeker**

Besides, if the hider is being chased by the seeker but cannot observe the seeker,

we also defines an attribute **isBeingChased** which will helps hider to realize that it is still being chased by the seeker, and choose the best move for the next step

## 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.

# User Interface and Game Play

A screenshot of a game

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

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedAfter 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.

A screenshot of a computer game

Description automatically generatedA screenshot of a computer game

Description automatically generatedOnly 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 (in level 3 or more, the hiders can make the seeker chased endlessly). And the begin button will be changed to "Restart", letting the user load the map again for another run.

# Evaluation

# Refs

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