HO CHI MINH UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

REPORT: Project 01: Hide and Seek

By:
22127057 – Đỗ Phan Tuấn Đạt
22127064 – Phạm Thành Đạt
22127123 – Lê Hồ Phi Hoàng
22127131 – Trần Nguyễn Minh Hoàng

A.TABLE OF CONTENTS

B.INTRODUCTION

This project was initialized as part of the course CSC14003 – Introduction to Artificial Intelligence. Its purpose is to help the students learn the basics of creating and implementing artificially intelligent agents through the simple childhood game hide-and-seek.

Note that the agents in this project are in no way capable of learning since all utilized algorithms are search algorithms, not neural networks.

C.WORKING ENVIRONMENT

I. Programing language

100% Python

II. Code editor

Visual Studio Code

III. Code management

GitHub

D. USER INSTRUCTIONS

- Simply run the outermost main.py file (NOT the main.py files in the level folders) and start clicking buttons.
- The processing before initializing the pathfinding can take a while, so do give it a second (up to 1 minute) to load.
- For more detailed instructions, watch here: https://www.youtube.com/watch?v=tou3pcZFlig

E. GENERAL CODE ANALYSIS

I. Seeker Agent

1. Code representation

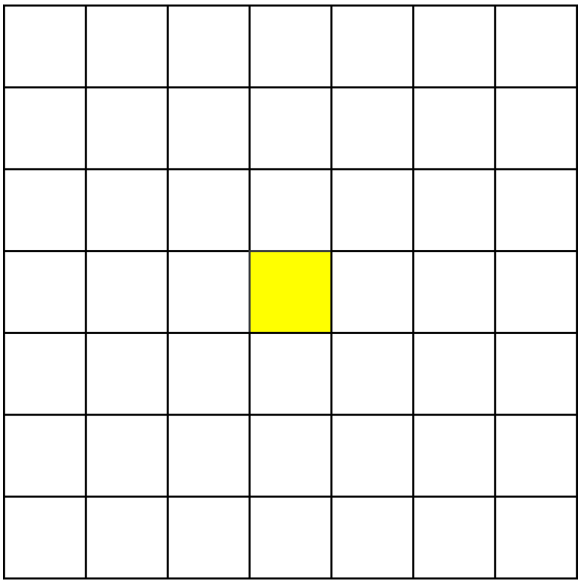
- The seeker agent is represented as the Seeker class, stored in the seeker.py file of each level.
- A Seeker object is constructed as follow:

```python
class Seeker:
    def __init__(self, map2d, seeker_pos, vision_range = 3, visionCount = 0, path_cost = 0, heuristic_cost = 0, parent = None, unobserved = None):
        self.map = Map(map2d.row, map2d.col, map2d.step, map2d.timeSignal)
        self.map.map = [row.copy() for row in map2d.map]
        self.map.obstacles = map2d.obstacles
        self.map.hider_pos = map2d.hider_pos
        self.map.hider_signal_pos = map2d.hider_signal_pos
        self.seeker_pos = seeker_pos
        self.vision_range = vision_range
        self.visionCount = visionCount
        self.path_cost = path_cost
        self.heuristic_cost = heuristic_cost
        self.parent = parent
        self.observed = set()
        self.unobserved = unobserved
        if unobserved == None:
            self.unobserved = set()
            for i in range(self.map.row):
                for j in range(self.map.col):
                    if self.map.map[i][j] == 0:
                        self.unobserved.add((i, j))
```

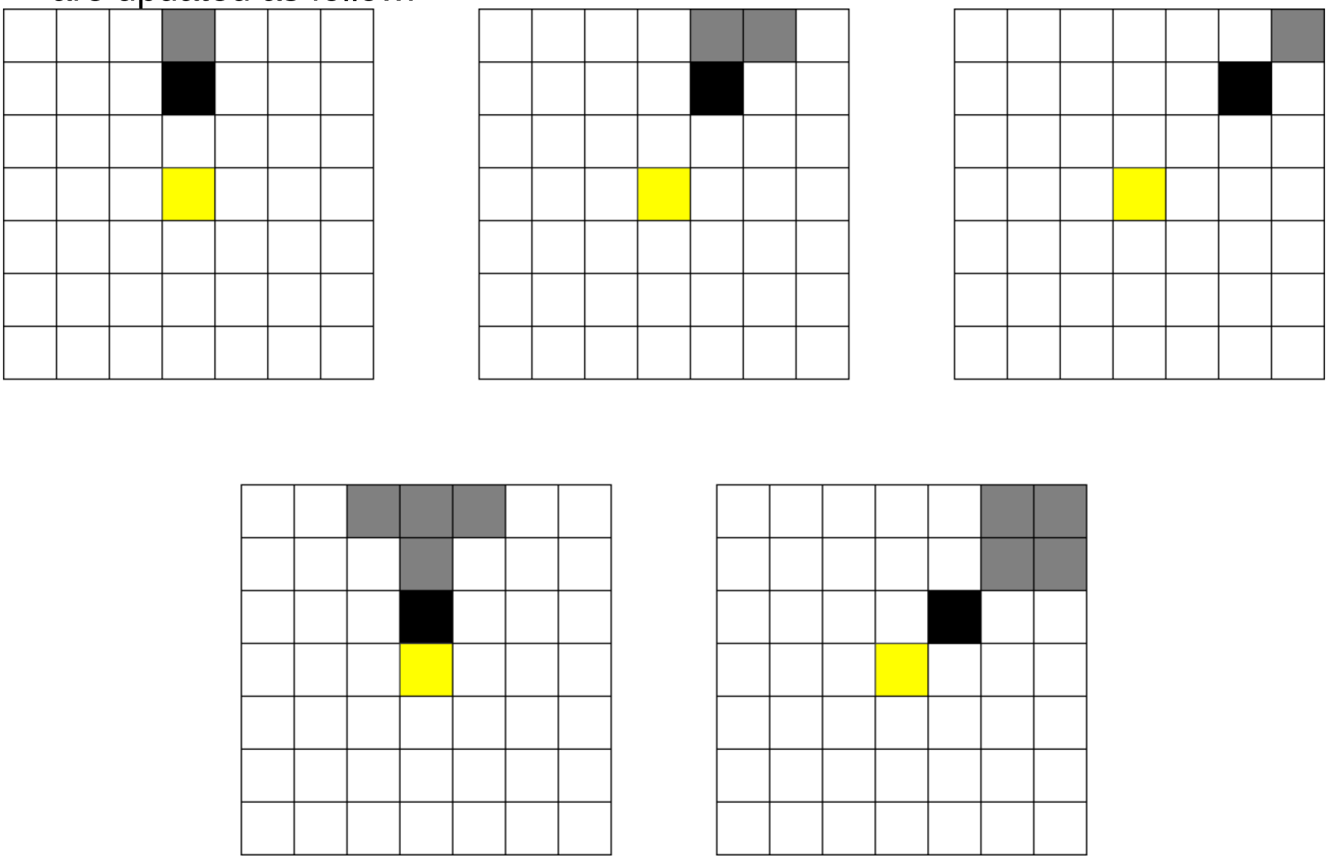
- In case you haven't realized, this Seeker class also serves a dual purpose as a representation of the puzzle state, hence the *path_cost*, *heuristic_cost* and *parent* paremeters used in initialization, as well as the *__lt__* operator implemented further down the source code. This may look a bit odd, but it will come in handy later.

2. Seeker's vision

- As required, the seeker has a vision that ranges 3 tiles away from it in all 8 directions, as illustrated below:



- When there is an obstacle/wall within its range of vision, observable tiles are updated as follow:



(**Black tiles** represents obstacles/walls; **grey tiles** are those that are now unobservable. The same logic applies for other quarters within the range of vision)

- In cases of long walls or multiple obstacles, the tiles that become unobservable are simply combinations of the above conditions.
- This is achieved by iterating over the obstacles on the map and calls the *blockVision* function to update the map, blocking visibility around obstacles.

```python
def blockVision(self, obstacle_pos):
        self.blockVisionVertical(obstacle_pos)
        self.blockVisionHorizontal(obstacle_pos)
        self.blockDiagonal(obstacle_pos)

```

- Seeker's vision is updated after every time the seeker takes a step.

3. Path-finding using search algorithms

- Within the Seeker class, 3 search algorithms are implemented:
    - Breath-first search (BFS)
    - Hill climbing search, in which the seeker tries to move towards positions with the highest possible number of observable tiles.
    - A* search using Manhattan distance between current position and target position as the heuristic function.
- The priorities of each search strategies are as follow:
    - When neither hider or signal is present within observable tiles, traverse the map using Hill climbing search.
    - Upon encountering a local maximum (when all possible moves lead to a reduce in observable ties), switch to BFS to search for another unexplored portion of the map.
    - At any point during traversal, if a signal or hider appears in sight, immediately switch to A* search to reach the position of said hider/signal.
    - While the seeker is approaching a specific signal, if a hider appears, it will change it targets to minimize steps taken to capture the hider.
- The seeker takes 1 step per time unit, and also remembers tiles it has already observed/walked through so as to reduce redundant steps.

II. Mapping

- The map in which the seeker and hider(s) will play in is represented by the Map class in the map.py file of each level.

```python
class Map:
    def __init__(self, row = 0, col = 0, step = 0, timeSignal = 5, hider_radius = 3):
        self.row = row
        self.col = col
        self.map = []
        self.obstacles = []
        self.step = step
        self.timeSignal = timeSignal
        self.hider_pos = None
        self.hider_signal_pos = None
        self.hider_radius = hider_radius
```

- The only way to input a map is though text (.txt) files, which must be formatted as requested by educators:

- The first line contains two integers N x M, which is the size of the map.
- N next lines represent the N x M map matrix. Each line contains M integers. The number at row i, column j determines whether wall, hiders or seeker is set. If there is wall at this position, we will have value 1. If there is hider, we will have value 2. If there is seeker, we will have 3. Otherwise (empty path), we will have 0.
- The last lines store 2 pair numbers indicate top left, bottom-right of each obstacle.

- Map class also offers multiple functions to assist with the operation of the program and debugging.
- Among them is *get_hider_pos()* function, which helps with locating hider(s). This function returns a single tuple representing the x, y coordinates of the hider in level 1, and a list of tuples containing the positions of all hiders present on the map in later levels.
- Similarly, in level 1, the *hider_pos* attribute was a single tuple representing the position of the hider. In level 2 and 3, *hider_pos* is a set containing multiple tuples representing the positions of multiple hiders.
- The program does not have a separate Hider class, so it is necessary to keep track of the position and status (captured or not) of the hider(s) on the map, with the assistance of the functions in main.py.

III. Graphical demonstration

This process is greatly assisted by the **pygame** module. So make sure to have the module installed before running the program.

1. Main menu

- Upon starting the program, users will be met with the main menu containing the title and subtitle of the game, along with some buttons.
- The buttons are objects of the Button class, which is designed specifically around the **pygame** module.

```python
class Button():
    def __init__(self, pos, image, text, fontSize, textColor, hoverSize):
        self.pos = pos
        self.image = image
        self.fontSize = fontSize
        self.font = pygame.font.Font(None, fontSize)
        self.text = text
        self.textImage = self.font.render(self.text, True, textColor)
        if self.image is None:
            self.image = self.textImage
        self.textColor = textColor
        self.rect = self.image.get_rect(center = (self.pos[0], self.pos[1]))
        self.textRect = self.textImage.get_rect(center = (self.pos[0], self.pos[1]))
        self.hoverSize = hoverSize
```

- Each button can change size when the cursor hovers on it. This is done by updating both the size of the text and background image of the button.

```python
def changeSize(self, mousePos):
    if (mousePos[0] > self.rect.left and mousePos[0] < self.rect.right
        and mousePos[1] > self.rect.top and mousePos[1] < self.rect.bottom):
        self.image = pygame.transform.scale(self.image, self.hoverSize)
        self.rect = self.image.get_rect(center = (self.pos[0], self.pos[1]))
        self.font = pygame.font.Font(None, self.fontSize + 3)
    else:
        self.image = pygame.transform.scale(self.image, self.rect.size)
        self.rect = self.image.get_rect(center = (self.pos[0], self.pos[1]))
        self.font = pygame.font.Font(None, self.fontSize)
```

- Upon clicking a button, it initiates a function, which opens another screen.

2. Demonstrating the running process

- The seeker's path will be illustrated in a "game runner" screen. This screen will initialize after all calculation and processing are done (meaning only after the seeker finishes the game) so it might take a while to show up.
- Since the whole path taken by the seeker is saved, the GUI will only have to display every step in a slideshow.

3. Score display

- The bottom of the game runner screen will have a bit of space reserved for score display.
- Score starts at 0, and decreases by 1 per time unit that passes. For each hider the seeker manages to catch, the score will increase by 20.
    - The *get_hider_pos()* function provided my Map class will help with indicating a capture.
- When terminating the game runner, said score will also be printed out to console for debugging.

F. LEVEL-SPECIFIC ANALYSIS

I. Level 1

- In this level, things are at their most simple form: 1 hider which cannot move, 1 seeker and no time limit; the game ends when the seeker successfully catch the hider (i.e. they occupy the same tile).
- The seeker's strategy is also at its base form that I have described in section C above.
- Every 5 time units, the hider will randomly place a signal on a single tile within its range of 3 tiles.

II. Level 2

- There are multiple stationary hiders in this level, the number of which the seeker knows.
- The strategy used by the seeker is largely the same, the difference being that it will not stop until all hiders are caught.
- Moreover, the change in data type of *hider_pos* means that instead of directly assigning the hider positions to *map2d.hider_pos*, the main function iterates over the *hider_pos_list*, creates a dictionary for each hider containing its position and potential signal area, and appends it to *list_potential_hider_signal*. The *hider_pos* is added to *map2d.hider_pos* inside the loop.

```
list_potential_hider_signal = []
    current_signal = set()
    for hider_pos in hider_pos_list:
        hider = {
            "hider_pos": hider_pos,
            "potential_signal": map2d.potentialSignalArea(hider_pos)
        }
        list_potential_hider_signal.append(hider)
        map2d.hider_pos.add(hider_pos)
        current_signal.add(random.choice(hider["potential_signal"]))
```

- Note that all of *list_potential_hider_signal*, *hider_pos* attribute in Map and *current_signal* are immutable objects which means they're pass by reference. This means that functions that have to do with map must return a new *current_signal* set as they are randomized once every 5 steps.

```
original_signal = current_signal.copy()
    for i in range(len(path)):
        if path[i].map.step >= path[i].map.timeSignal:
            for signal in original_signal:
                if signal == path[i].seeker_pos:
                    pass
                elif signal in path[i].observed:
                    path[i].map.map[signal[0]][signal[1]] = 4
                else:
                    path[i].map.map[signal[0]][signal[1]] = 0
            if path[i].map.step % path[i].map.timeSignal == 0:
                current_signal = set()
                for hider in list_potential_hider_signal:
                    if hider["hider_pos"] in path[i].map.hider_pos:
                        signal = random.choice(hider["potential_signal"])
                        while signal == path[i].seeker_pos:
                            signal = random.choice(hider["potential_signal"])
                        current_signal.add(signal)
            for signal in current_signal:
                path[i].map.map[signal[0]][signal[1]] = 5
            path[i].map.map[path[i].seeker_pos[0]][path[i].seeker_pos[1]] = 3
```

- This is why it is necessary to create *original_signal* which copies the *current_signal* so that we can update the map without it changing every loop.

```
signal_pos = path[0].map.hider_signal_pos
    for i in range(len(path)):
        if path[i].map.step >= path[i].map.timeSignal:
            signal = path[i].map.hider_signal_pos
            if signal == path[i].seeker_pos:
                pass
            elif signal in path[i].observed:
                path[i].map.map[signal[0]][signal[1]] = 4
            else:
                path[i].map.map[signal[0]][signal[1]] = 0
            if path[i].map.step % path[i].map.timeSignal == 0:
                signal_pos = random.choice(potentialSignalArea)
                while signal_pos == path[i].seeker_pos:
                    signal_pos = random.choice(potentialSignalArea)
            if path[i].seeker_pos != signal_pos:
                path[i].map.map[signal_pos[0]][signal_pos[1]] = 5
            path[i].map.hider_signal_pos = signal_pos
```

- If we had implement the same as level 1, signal_pos would change after every single loop, making it virtually impossible to know where the original signal actually is to remove, causing multiple signals to appear at times.

III. Level 3

- Level 3 is mostly similar to level 2, however, there's a slight difference in *encounterHider* function. This is because, after seeker saw a hider, it moves 1 step closer to a hider and approach said hider's 2-tile range of vision. Then the *chaseDownHider* function is called.
- Here is how it works:

```python
while True:
        valid_moves = checkValidMovesforHider(result.map.map, hider_pos)
        for move in valid_moves:
            if move == result.seeker_pos or move in result.map.hider_pos:
                valid_moves.remove(move)
        if valid_moves != []:
            new_pos = random.choice(valid_moves)
        else:
            new_pos = hider_pos
        results = result.generateNewStates()
        mindistance = 1e9
        next_pos = None
        for seeker in results:
            if seeker.seeker_pos == new_pos:
                next_pos = seeker
                break
            distance = calcUclidianDistance(seeker.seeker_pos, new_pos)
            if distance < mindistance:
                mindistance = distance
                next_pos = seeker
        result = next_pos
        result.map.map[hider_pos[0]][hider_pos[1]] = 4
        for signal in original_signal:
            if signal == result.seeker_pos:
                pass
            elif signal in result.observed:
                result.map.map[signal[0]][signal[1]] = 4
            else:
                result.map.map[signal[0]][signal[1]] = 0
        hider_pos = new_pos
        result.map.map[hider_pos[0]][hider_pos[1]] = 2
        result.map.map[result.seeker_pos[0]][result.seeker_pos[1]] = 3
        if result.seeker_pos == hider_pos:
            break
    return result
```

  - The function enters a loop where it repeatedly calculates valid moves for the hider. It ensures that the hider does not move onto the seeker's position or onto another hider's position.
  - If there are valid moves available for the hider, it randomly chooses one of them if not then it will stand still and not move.
  - It then generates new states of the seeker after the hider's move and calculates the Euclidean distance between each potential seeker position and the new hider position. This is a new Heuristic function that is applied only in level 3.
  - The function selects the seeker position with the minimum distance to the new hider position.

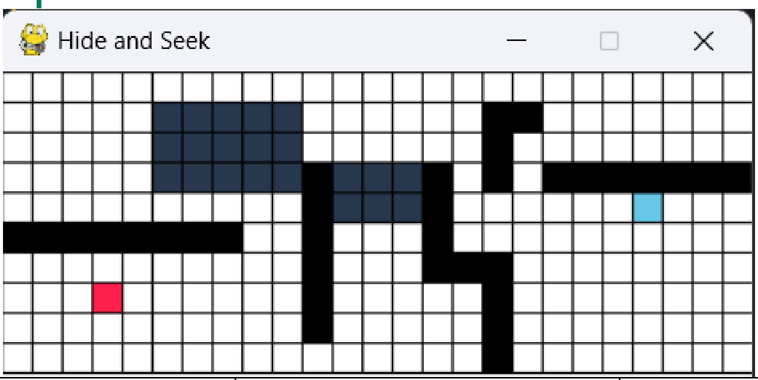
- It updates the map with the new positions of the seeker and the hider.
- The loop continues until the seeker captures the hider by reaching its position.
- Once the seeker captures the hider, the function breaks out of the loop and returns the final state of the seeker.

G. EVALUATION & COMMENTS

- This section will focus on evaluating the performance of the program. Each level features 5 different maps. 3 runs will be performed on each map to record the average time and resource consumed by the program.

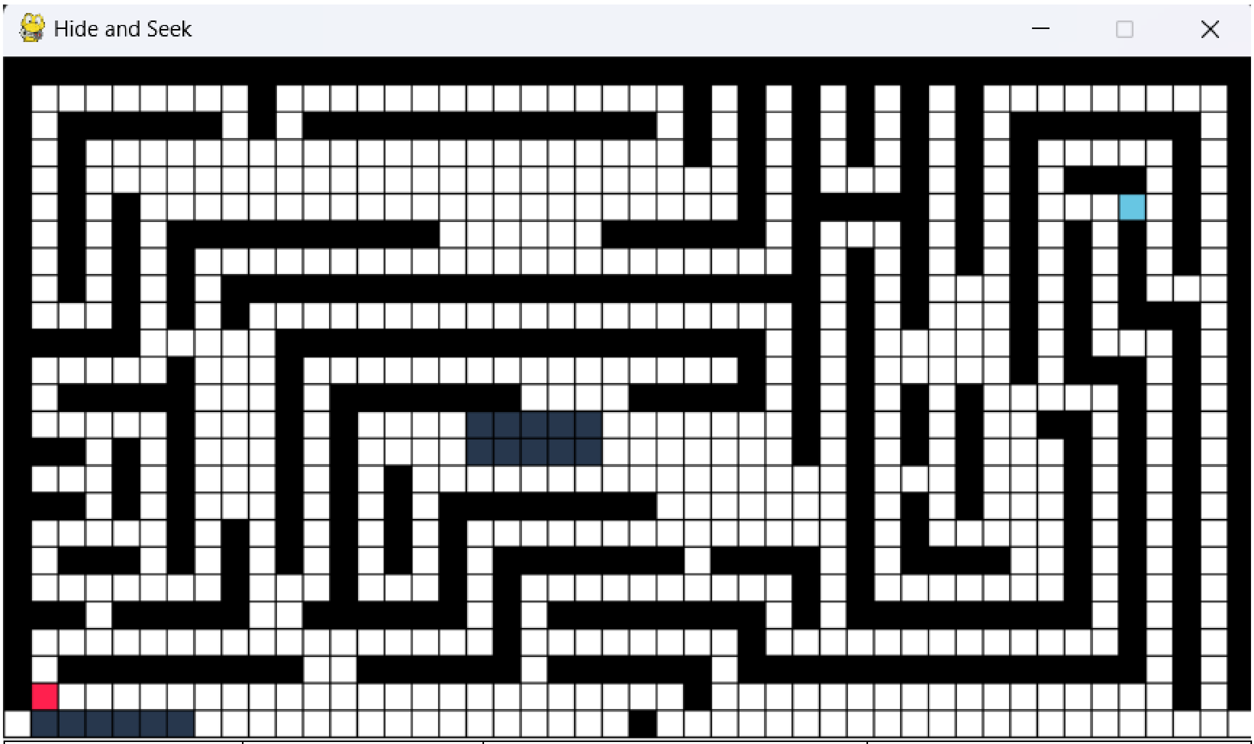
I. Level 1

1. Map 1



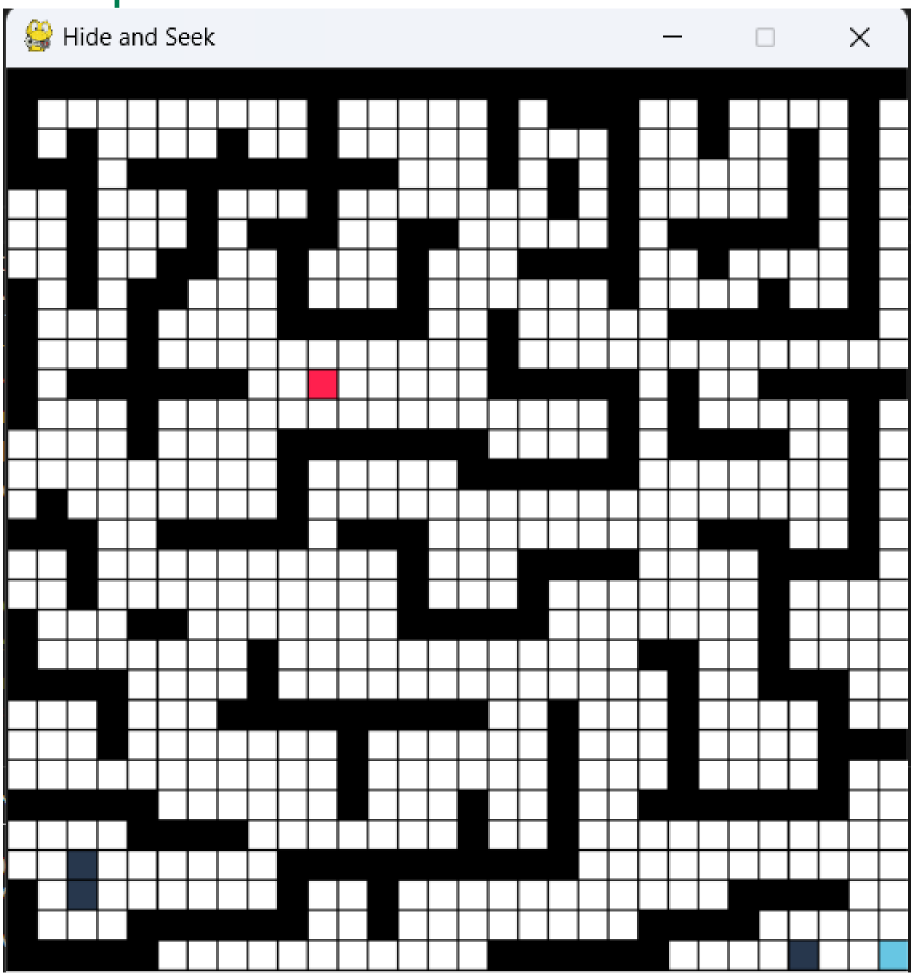
Run	Score	Time (miliseconds)	Space (megabytes)
1	-13	22.105455	1.148438
2	-28	95.970869	3.605469
3	-13	9.648561	1.164062
Avr	-18	42.57496167	1.972656333

2. Map 2



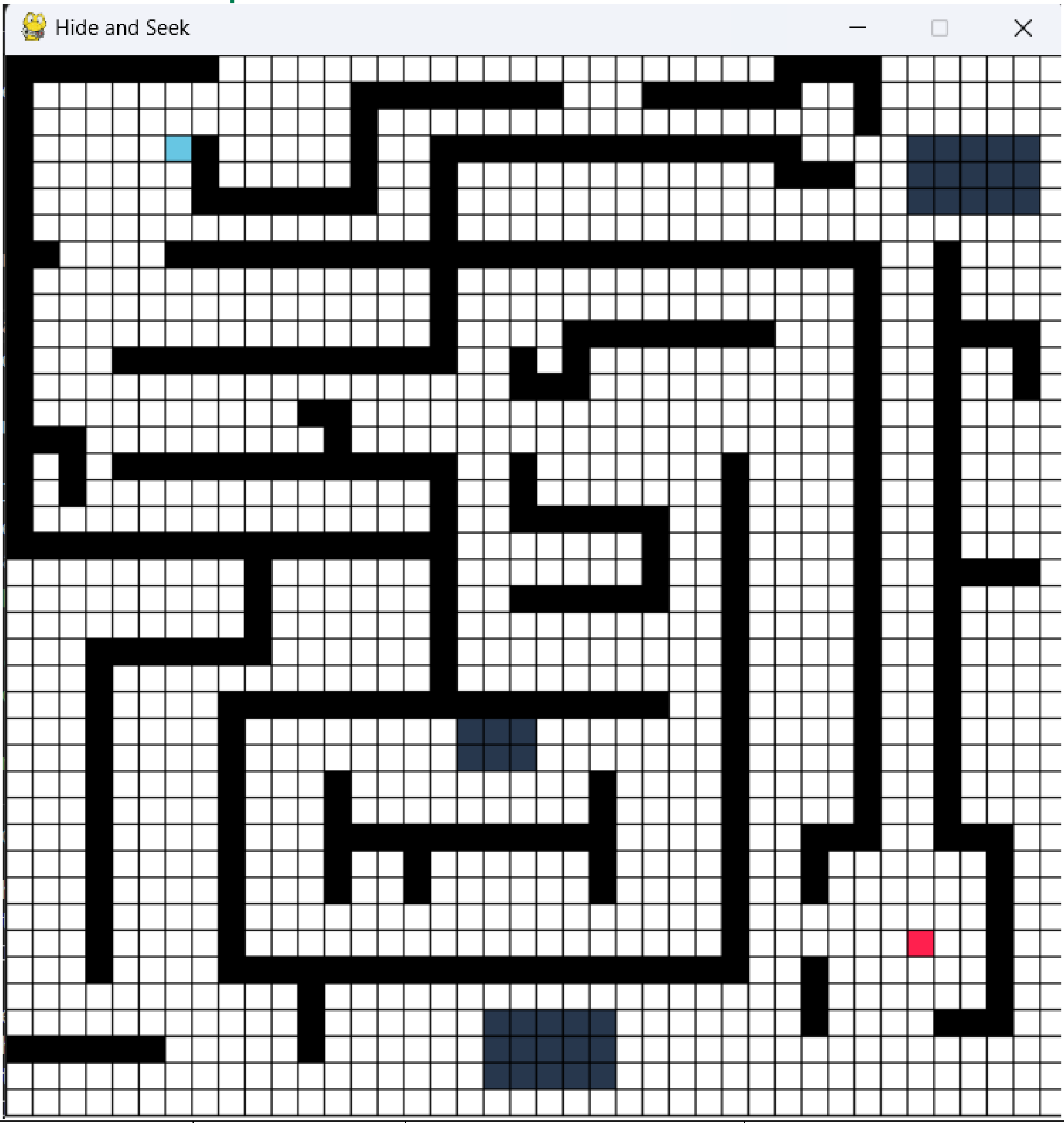
Run	Score	Time (miliseconds)	Space (megabytes)
1	-422	2164.680	30.094
2	-47	13.485	1.484
3	-333	1294.753	17.547
Average	-267.3	1157.64	16.375

3. Map 3



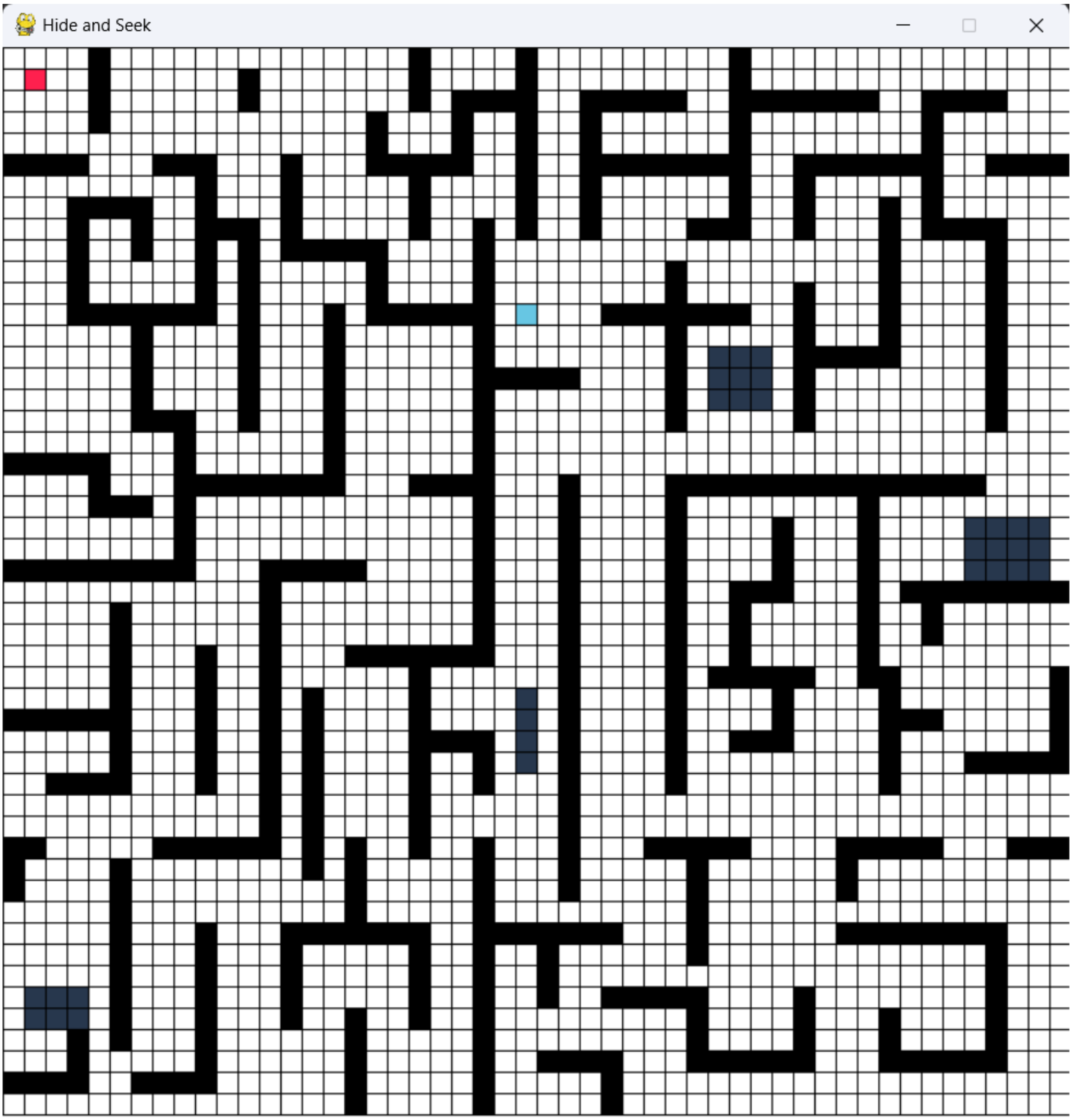
Run	Score	Time (miliseconds)	Space (megabytes)
1	-209	2418.203	26.703
2	-165	1460.073	16.918
3	-192	1903.081	26.387
Average	-188.67	1927.110	23.336

4. Map 4



Run	Score	Time (miliseconds)	Space (megabytes)
1	-91	642.522	14.047
2	-63	174.661	10.535
3	-132	3776.902	18.035
Average	-95.33	1531.36	114.205

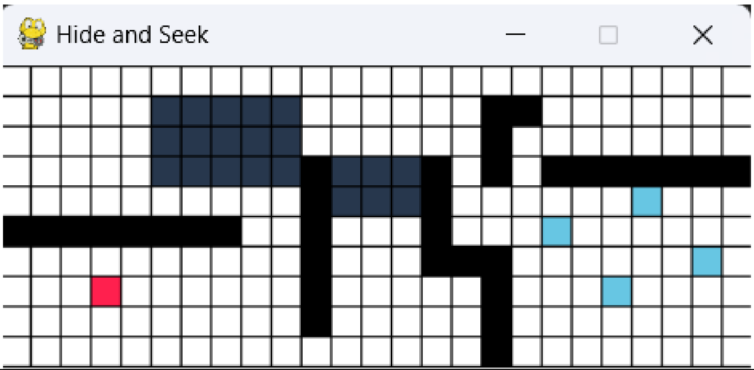
5. Map 5



Run	Score	Time (miliseconds)	Space (megabytes)
1	-10	26.582	3.504
2	-10	14.787	3.547
3	-11	85.644	9.652
Average	-10.33	42.447	5.5677

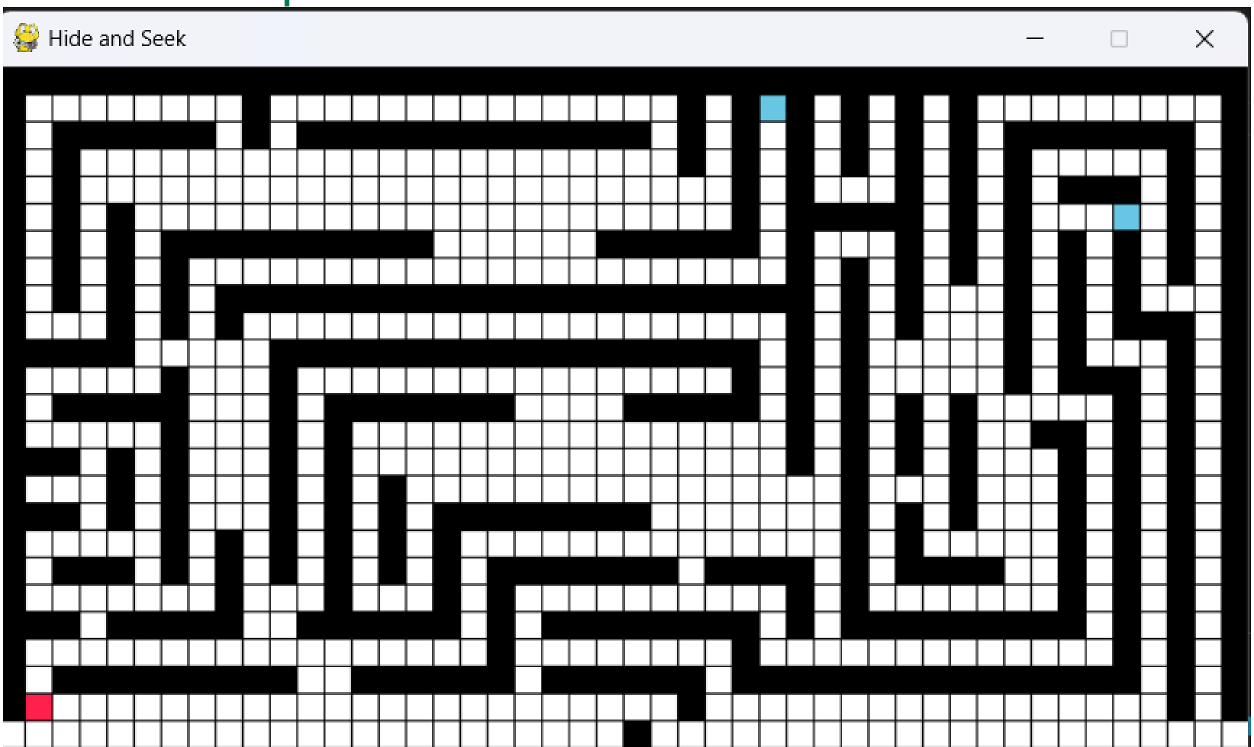
II. Level 2

1. Map 1



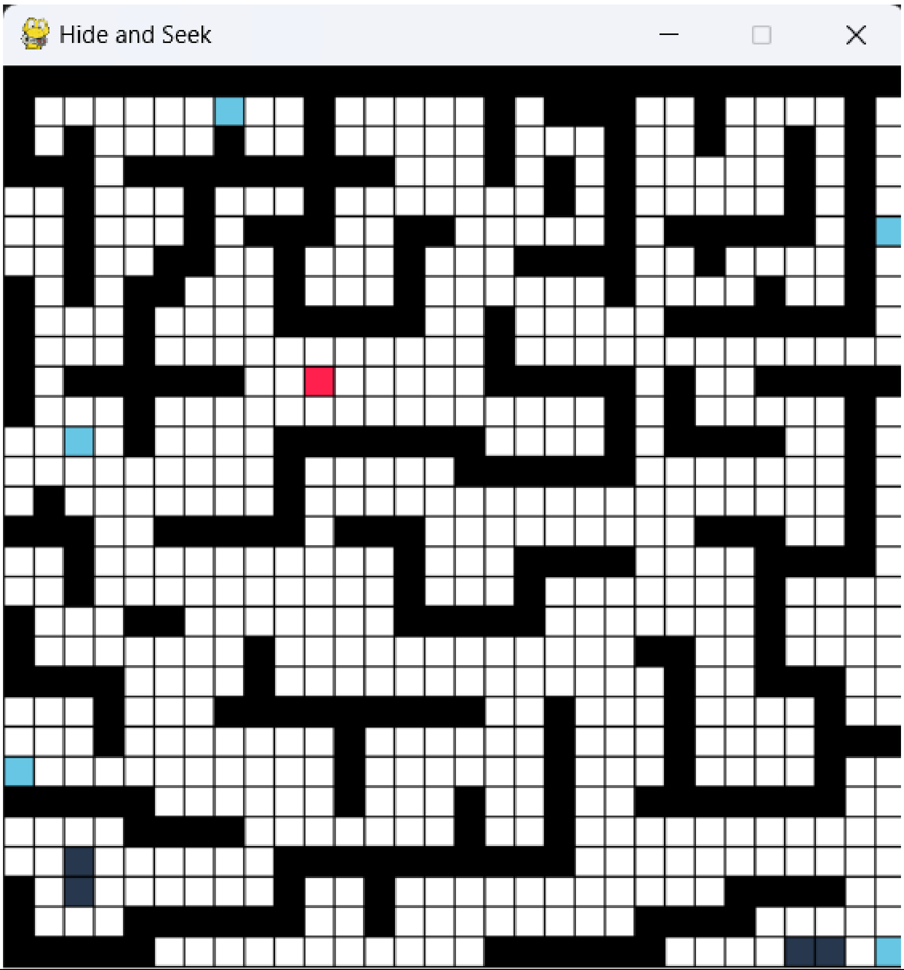
Run	Score	Time (miliseconds)	Space (megabytes)
1	27	297.400	3.254
2	35	70.783	1.465
3	35	230.805	2.758
Average	32.33	199.66	2.492

2. Map 2



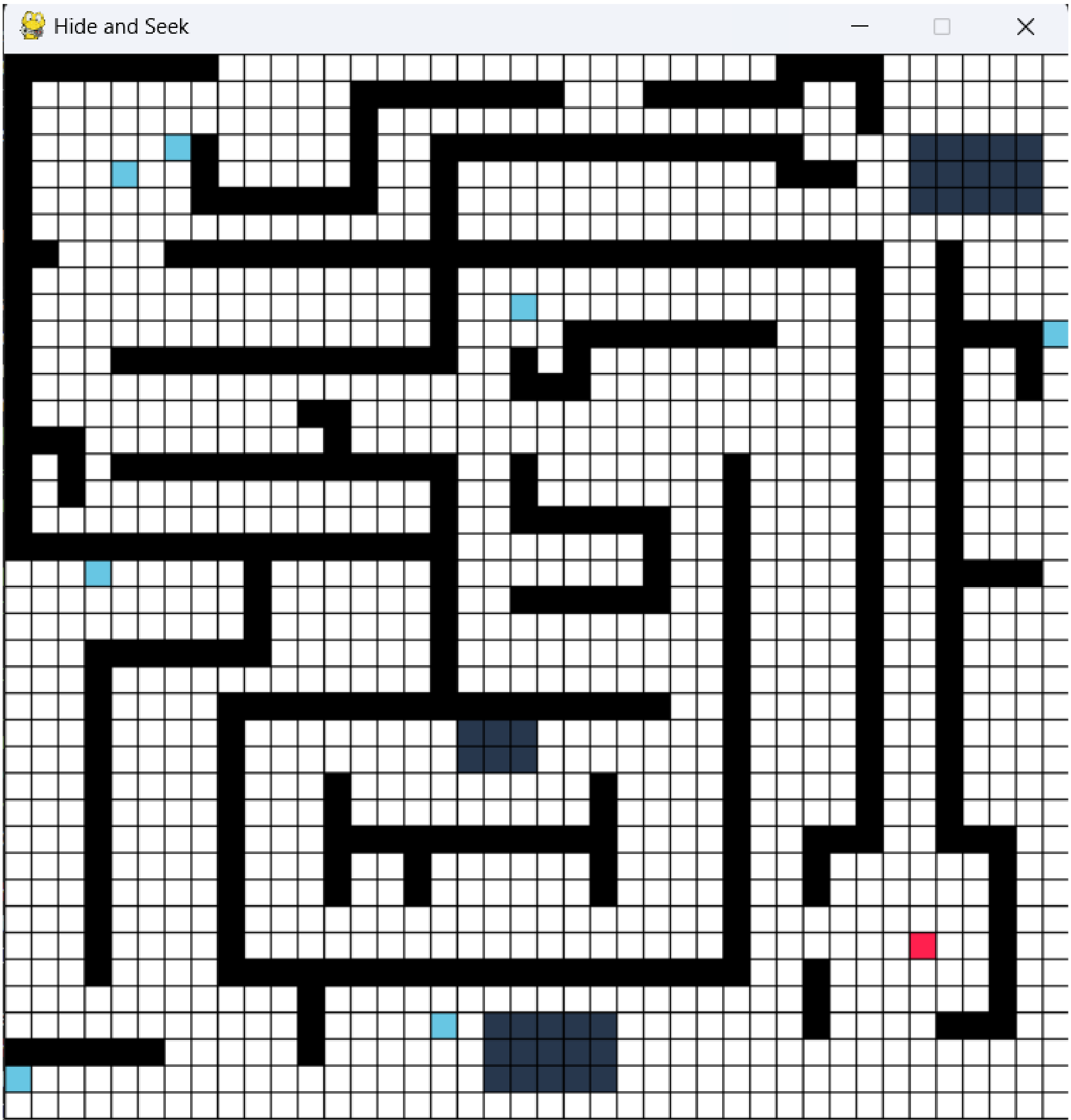
Run	Score	Time (miliseconds)	Space (megabytes)
1	-443	9585.526	36.570
2	-442	10845.965	38.926
3	-356	8593.667	29.031
Average	-413.66	9675.052	34.842

3. Map 3



Run	Score	Time (miliseconds)	Space (megabytes)
1	-184	10921.771	29.594
2	-85	8108.374	34.168
3	-217	11707.010	26.641
Average	-162	10245.718	30.134

4. Map 4



Run	Score	Time (miliseconds)	Space (megabytes)
1	-172	15674.640	48.75
2	-215	16618.833	52.109
3	-338	53120.585	118.676
Average	-241.6	28471.352	73.178

5. Map 5

Run	Score	Time (miliseconds)	Space (megabytes)
1	-470	118617.489	160.703
2	-529	115374.011	112.895
3	-355	66782.402	109.594
Average	-451.33	100257..9673	127.73

III. Level 3

- Level 3 uses the same set of maps as level 2, so only the result tables will be listed
    - Map 1

Run	Score	Time (miliseconds)	Space (megabytes)
1	40	34.73	1.25
2	30	38.686	1.051
3	35	69.499	1.547
Average	35	47.638	1.283

- Map 2

Run	Score	Time (miliseconds)	Space (megabytes)
1	-168	4028.924	24.477
2	-278	6556.450	30.910
3	-299	10329.924	32.977
Average	-248.33	6971.766	29.454

- Map 3

Run	Score	Time (miliseconds)	Space (megabytes)
1	-186	13528.262	37.375
2	-197	10723.233	25.934
3	-193	12502.166	28.02
Average	-192	12251.22	30.443

- Map 4

Run	Score	Time (miliseconds)	Space (megabytes)
1	-157	12374.505	50.457
2	-129	15581.947	67.988
3	-337	57083.051	159.547
Average	-207.66	28346.501	92.664

- Map 5

Run	Score	Time (miliseconds)	Space (megabytes)
1	-565	181661.581	297.09
2	-479	96586.793	166.895
3	-546	131795.551	123.34
Average	-530	136681.308	199.775

IV. Comments

From the data gathered, there are a few comments that can be made:

- Obviously, larger maps will lead to lower score due to more wandering around. Additionally, the processing takes substantially more time and resources the larger the map, and the further the hider(s) are from the seeker.
- Smaller maps with more hiders in them will naturally lead to higher scores thanks to the reward/penalty rate of 20/1.
- Escaping hiders are surprisingly easy to catch in level 3, which might be because of how we programmed the hiders to run.
- Signals sometimes becomes distractions, leading the seeker away from the hider, or to the wrong side of a wall.

H.ASSIGNMENT PLAN

Student ID	Full name	Assigned task	Completion evaluation	Total contribution
22127057	Đỗ Phan Tuấn Đạt	- Design and implement main menu + selection screen UI. - Write analysis for above sections + comments on experiment results. - Combine and design final report.	100%	25%
22127064	Phạm Thành Đạt	- Design and implement algorithms for seeker's vision and hider's random signaling. - Write analysis for above sections. - Film user instruction video and design input maps.	100%	25%
22127123	Lê Hồ Phi Hoàng	- Design and implement UI for displaying the seeker's route. - Implemented required mapping input into Map class. - Write analysis for above sections.	100%	25%
22127131	Trần Nguyễn Minh Hoàng	- Design and implement algorithms for seeker's map traversal and chasing hider. - Write analysis for above sections. - Carried out experiments for evaluating and debugging program.	100%	25%

I. ESTIMATED COMPLETION

No.	Specifications	Scores	Estimated completion
1	Finish level 1 successfully.	15%	100%
2	Finish level 2 successfully.	15%	100%
3	Finish level 3 successfully.	15%	100%
4	Finish level 4 successfully.	10%	0%
5	Graphical demonstration of each step of the running process. You can demo in console screen or use any other graphical library.	10%	100%
6	Generate at least 5 maps with difference in number and structure of walls, hiders, seeker, and obstacles.	5%	100%
7	Report your algorithm, experiment with some reflection or comments.	30%	100%

J. REFERENCES

- https://www.youtube.com/watch?v=GMBqjxcKogA
- *Artificial Intelligence: A Modern Approach - 3rd Edition* by Stuart Russel and Peter Norvig
- https://www.geeksforgeeks.org/a-search-algorithm/
- https://www.codecademy.com/resources/docs/ai/search-algorithms/hill-climbing
- https://datascience.stackexchange.com/questions/20075/when-would-one-use-manhattan-distance-as-opposed-to-euclidean-distance