

AI Capstone HW3

Minesweeper

110550088 李杰穎

May 7, 2023

In this homework, we're asked to write a minesweeper game and use logical inference to build an AI that automatically play minesweeper game.

I wrote this homework using Python, in this report, I will first introduce the modules I create for this homework including module for literal, clause, knowledge base, game and player. And then talk about how to use logical inference technique, especially in this homework, resolution, to develop a AI for this game.

1 Game Module

The code of Game Module can be referred to the appendix, Code 1. In this module, I implement some useful functions that will be later used in the development. I think the comments in code are clear enough to explain the code, therefore, I will not make further explanation regarding to this module. Noted that the `open_cell` function means that marked that cell with mines or safe.

2 Logical Related Module

In this homework, we will need to use logical inference to solve the minesweeper game, thus we need to create modules that deal with logic. On the top of everything, we have a knowledge base (KB), which contains a set of logical statements or facts expressed in a formal language. These statements are represented using clauses, which are composed of a disjunction of literals.

For example, we could represent the fact that a cell is adjacent to a mine using a clause such as $\neg A \vee B$ where A represents the safe cell and B represents the mine. We could also represent the fact that a cell does not contain a mine

using a clause such as $(\neg A)$, where \neg represents negation.

Literals are the building blocks of clauses and represent a proposition that is either true or false. In the context of the minesweeper game, we can represent each cell on the game board using a propositional symbol, where the symbol represents the proposition that the cell contains a mine or not. For example, "A" could represent the proposition that cell A contains a mine, while " $\neg A$ " represents the proposition that cell A does not contain a mine.

By using logical inference on the knowledge base and the rules of the game, we can deduce the state of the cells on the game board and determine which cells contain mines. This involves using techniques such as resolution to draw logical conclusions based on the knowledge base.

2.1 Literal Module

The code can be referred to Code 2. Each literal contains two parts, the first part is the position of a certain cell, stores as a tuple, and a boolean `posi` indicates that whether this literal has negation or not. And because we will later store the literal in a set, we will need to define the hash function for any literal, in this homework, the hash function for literal is hash value of its string representation, the negation is indicated by a prime (''). For example, the literal of cell at position (x, y) with negation is (x, y)'

2.2 Clause Module

The code for this module can be referred to Code 3. This module defines a Clause class that used a set to store literals in the clause. A clause is a disjunction of literals, where a literal is a basic proposition that is either true or false.

In this module, I defined some useful function, like `str`, which defined how to print a clause to console, and `eq` function, defined the equity of two clauses.

2.3 Knowledge Base Module

The code of this module can be referred to Code 4. This module defines a **KB** class that represents a knowledge base, which is a collection of logical clauses. The knowledge base is used to store knowledge about a domain of interest and to perform logical inference to draw conclusions from that knowledge.

The `init` method initializes the **KB** object with a set of clauses. The `insert` method is used to insert a new clause into the knowledge base, while ensuring that the knowledge base remains consistent and does not contain redundant or contradictory information.

The `insert` method takes two parameters, a Clause object to be inserted and the **KB0** object that contains the inferred clauses. The method first apply resolution to all clauses in **KB0**, noted that **KB0** stores all the clauses that has been already inferred and every clause is single-literal clause. This step decrease the number of literal in inserted clause. Then, it checks if the clause is already in the knowledge base, or if it is a superset of another clause in the knowledge base. If either of these conditions is true, the method returns without modifying the knowledge base. Otherwise, the clause is added to the knowledge base. And if any of clause in **KB** is a superset of inserted clause, then remove that clause.

The **KB** class is useful for representing and manipulating knowledge bases in logical inference systems. The `insert` method ensures that the knowledge base remains consistent and that new clauses are added in a way that preserves the logical structure of the knowledge base.

3 Player Module

The code of this module is Code 5. This module implements the AI that automatically play the minesweeper game, and inserted the clauses to **KB** by the given hint.

3.1 Generate Clauses by Given Hint

A hint is made up of the positions of unmarked cell around the opened cell and the number of mines in those unmarked cell. Suppose the positions are x_1, x_2, \dots, x_m and there are n mines in those m unmarked cell.¹ The inserted clauses are generated as follow:

- If $n = m$, then all the unmarked cells are mines. Therefore inserted (x_i) where $i = 1, 2, \dots, m$ to **KB**.
- If $n = 0$, then all the unmarked cells are safe. Therefore inserted $\neg(x_i)$ where $i = 1, 2, \dots, m$ to **KB**.
- If $m > n > 0$, which is the general case, inserted two types of clauses to **KB**.
 1. Inserted $\binom{m}{m-n+1}$ clauses, each clause contains $m - n + 1$ positive literal.
 2. Inserted $\binom{m}{n+1}$ clauses, each clause contains $n + 1$ positive literal.

The above results are given by pigeonhole theorem, for the first type, because there are only $m - n$ safe cells, for any clause that contains more than $m - n$ literals, it must have at least one literal that corresponds to mine cell, thus the whole clause will be true. Similarly, for the second type, there are only n mine cells, for any clause with more than n literals, it must contain at least one literal that corresponds to safe cell, thus the whole clause will be true. These two type of clauses are the essential of this logical inference, without those clauses, we can't get the correct results.

Noted that when inserting clause into **KB**, we follow the process mentioned in subsection 2.3.

3.2 The Inference Process

The inference process is the main part of AI.

It first check if there are any single literal clause in **KB**. If there contains any, that called it C , it move C to **KB0**, which is the knowledge base that stores inferred clauses, and because C is single literal, which means we already know that the corresponded cell is a safe cell or mine. Thus, we can open the cell and get the hint if the cell is safe, then

¹ x_i is a tuple

inserting new clause by the above process. And then, we can apply matching to every other clause in **KB** with C . The matching process will be later described in subsection 3.3. Simply put, the matching process is to apply resolution to clauses and decrease the number of literal in clauses. Make the clause stricter.

If there doesn't contain any single literal clause, then apply pairwise matching to clauses in **KB**. These process is meant to generate single literal clauses eventually. If we can't get single literal clause for multiple iterations. Then the game is likely to be stuck, we can terminate the game play.

3.3 The Matching Process

The matching process involves two clauses, let's called them a and b . we want to simplify a and b using resolution and merged into one clauses c .

It first checked whether $a = b$ or $a \subset b$ or $b \subset a$. If any of conditions hold, we can only leave one stricter clauses. If none of the conditions hold, we can enter next step, which is applying resolution to two clauses to generate new clause. Noted that because we don't want to make **KB** grow to fast, thus if the number of literals of a and b are both greater than 2. Then we don't apply resolution in matching process.

4 Experiments and Results

In this section, I will mainly discuss about the performance of our game AI, and analysis it.

First of all, because this AI is based on logical inference., when opening a cell, the AI is sure about whether it's a safe cell or mine. It means that the AI will never be wrong, it will only have two outcome, win or stuck. Win means correctly marked every cell. Stuck means the we can't have further inference.

And also, in order to increase the running performance, the results in this section are executed in **PyPy**. PyPy is an alternative implementation of the Python programming

language that aims to be faster, more memory efficient, and more compatible with existing Python code than the standard CPython interpreter.

4.1 Performance of Different Difficulty

The board configuration of each difficulty is defined as in Table 1.

Table 1: The board configuration of each difficulty.

| Difficulty | Board Size | Number of Mines |
|------------|----------------|-----------------|
| Easy | 9×9 | 10 |
| Medium | 16×16 | 25 |
| Hard | 16×30 | 99 |

The results below is running on my personal computer with single thread program. For each difficulty, I tested 5 games, and each game is running independently.

4.2 Performance of Different Number of Initial Safe Cell

At the beginning of the game, the game module would first give the player the positions of some initial safe cell. The default value for the number of these safe cells is $\sqrt{\# \text{ of cell in board}}$. In this section, I will test how this number effect the running time for the program. And because running this experiments with hard difficulty is time-consuming, I will run the experiment with medium difficulty.

The default value of initial safe cell is 16. I will change this number to 8, 16, 32, 64, 128 and 231. And measure the average running time of 5 games.

Table 2: The performance with different number of init safe cell. The pairwise matching is the average number of running pairwise matching.

| Init. Safe Cell | Win | Stuck | Avg. Time | Std. Time | Pairwise Matching |
|-----------------|-----|-------|-----------|-----------|-------------------|
| 8 | 5 | 0 | 1.47 | 0.37 | 0.6 |
| 16 | 5 | 0 | 2.65 | 1.86 | 0.8 |
| 32 | 5 | 0 | 3.59 | 1.17 | 0 |
| 64 | 5 | 0 | 3.88 | 0.92 | 0 |
| 128 | 5 | 0 | 3.61 | 0.13 | 0 |
| 231 | 5 | 0 | 2.73 | 0.36 | 0 |

5 Conclusion and Future Work

In this section, I will mainly talk about the conclusion toward this homework, and some future work to improve the performance of this game AI.

First of all, this homework let me understand the concept of logical inference, and how it can apply to solve a real world problem. However, this method is quite time-consuming, because we need to apply resolution repeatedly to many clauses in knowledge base.

Therefore, for future work, I want to first improve the performance of my program. This program has much space to improve, including the data structure to store clauses and the matching process.

Appendix A Code of Modules and Functions

Code 1: Game Module

```

1 class Game:
2     def __init__(self, difficulty=0):
3         '''
4             Initialize the game board
5
6             Parameters
7             -----
8             difficulty : int
9                 0: Easy, 1: Medium, 2: Hard
10
11             Returns
12             -----
13             None
14             '''
15         board_configurations = [
16             (9, 9, 10), # Easy
17             (16, 16, 25), # Medium
18             (16, 30, 99) # Hard
19         ]
20         self.h, self.w, self.num_of_mines
21         ↪ = board_configurations[difficulty]
22         ↪ # height, width, number of mines
23         self.board = [[0 for _
24             ↪ in range(self.w)] for _ in range(self.h)]
25         ↪ # -1: mine, 0~8: number of mines around
26         self.shown_cell = [[False for _
27             ↪ in range(self.w)] for _ in range(self.h)]
28         ↪ # Indicate the cell is opened or not
29         self.mine_pos = set() # The position of mines
30
31         # Randomly generate mines
32         while
33             ↪ len(self.mine_pos) < self.num_of_mines:
34                 i = random.randrange(self.h)
35                 j = random.randrange(self.w)
36                 if (i, j) not in self.mine_pos:
37                     self.mine_pos.add((i, j))
38                     self.board[i][j] = -1
39
40     def open_cell(self, cell, safe):
41         '''
42             Open the cell
43             ↪ and return the number of mines around the cell
44
45             Parameters
46             -----
47             cell : tuple

```

```

40         The position of the cell
41         safe : bool
42         True
43         ↪ if the cell is safe, False if the cell is a mine
44
45     Returns
46     -----
47     int
48         The number of mines
49         ↪ around the cell, return -1 if wrongly opened
50         '''
51         if ((cell in self.mine_pos) ^ (not safe)):
52             ↪ or self.shown_cell[cell[0]][cell[1]]:
53                 return -1
54         if cell not in self.mine_pos:
55             self.board[cell[0]][cell[1]]
56             ↪ = self.get_surround_mines(cell)
57         else:
58             self.board[cell[0]][cell[1]] = "X"
59
60         self.shown_cell[cell[0]][cell[1]] = True
61
62         return self.board[cell[0]][cell[1]]
63
64     def get_hint(self, cell):
65         '''
66             Get the hint of the cell
67
68             Parameters
69             -----
70             cell : tuple
71                 The position of the cell
72
73             Returns
74             -----
75             List
76                 The list of the cells around the cell
77             int
78                 The number of mines around the cell
79             '''
80             cnt = 0
81             res = []
82             for i in range(cell[0]-1, cell[0]+2):
83                 for j in range(cell[1]-1, cell[1]+2):
84                     if i < 0 or i
85                     ↪ >= self.h or j < 0 or j >= self.w:
86                         continue
87                     if self.shown_cell[i][j]:
88                         continue
89                     if (i, j) != cell:
90                         if (i, j) in self.mine_pos:
91                             cnt += 1
92                         res.append((i, j))

```

```

88     return res, cnt
89
90     def get_surround_mines(self, cell):
91         '''
92         Get the number of mines around the cell
93
94         Parameters
95         -----
96         cell : tuple
97             The position of the cell
98
99         Returns
100         -----
101         int
102             The number of mines around the cell
103         '''
104         cnt = 0
105         for i in range(cell[0]-1, cell[0]+2):
106             for j in range(cell[1]-1, cell[1]+2):
107                 if (i, j) in self.mine_pos:
108                     cnt += 1
109         return cnt
110
111
112     def get_init_safe_cells(self):
113         '''
114         Get the initial safe cells
115
116         Parameters
117         -----
118         None
119
120         Returns
121         -----
122         set
123             The set of the initial safe cells
124         '''
125         num = round(math.sqrt(self.h * self.w))
126         # num = 10
127         init_cells = set()
128         while len(init_cells) < num:
129             i = random.randrange(self.h)
130             j = random.randrange(self.w)
131             if (i, j) not in self.mine_pos
132                 and (i, j) not in init_cells:
133                 init_cells.add((i, j))
134
135         return init_cells
136
137     def print_board(self):
138         '''

```

```

138         Print the game board. ? means the cell
139         ↪ is not opened yet. X means the cell is a mine.
140         ↪ 0~8 means the number of mines around the cell.
141
142         Parameters
143         -----
144         None
145
146         Returns
147         -----
148         None
149
150         '''
151         os.system('cls')
152         for i in range(self.h):
153             for j in range(self.w):
154                 if self.shown_cell[i][j]:
155                     print(self.board[i][j], end=' ')
156                 else:
157                     print('?', end=' ')
158             print()

```

Code 2: Literal Module

```

1 class Literal:
2     '''
3     A Literal is a cell with a positive or
4     ↪ negative sign. For example, (0, 0) is a positive
5     ↪ Literal, and (0, 0)' is a negative Literal.
6     '''
7     def __init__(self, cell, is_posi):
8         '''
9         Initialize the Literal
10
11         Parameters
12         -----
13         cell : tuple
14             The position of the cell
15         is_posi : bool
16             True if the Literal
17             ↪ is positive, False if the Literal is negative
18
19         Returns
20         -----
21         None
22
23         '''
24         self.cell = cell
25         self.posi = is_posi
26
27     def __eq__(self, other):
28         '''
29         Check if two Literals are the same

```

```

26     '''
27     return self.cell
28     ↪ == other.cell and self.posi == other.posi
29
30 def __str__(self):
31     '''
32     Return the string of the Literal
33     '''
34     return str(self.cell)
35     ↪ + (' ' if self.posi else '')
36
37 def __hash__(self):
38     '''
39     Return the hash value of the Literal
40     '''
41     return hash(str(self))

```

Code 3: Clause Module

```

32
33 def __len__(self):
34     '''
35     Return the number of Literals in the clause
36     '''
37     return len(self.literals)
38
39 def __hash__(self):
40     '''
41     Return the hash value of the clause
42     '''
43     return hash(str(self))
44
45 def __copy__(self):
46     '''
47     Return the copy of the clause
48     '''
49     return Clause(self.literals.copy())

```

Code 4: KB Module

```

1 class Clause:
2     '''
3     A clause is a set of literals
4     '''
5     def __init__(self, literals=[]):
6         '''
7         Initialize the clause
8
9         Parameters
10        -----
11        literals : List
12            The List of literals
13
14        Returns
15        -----
16        None
17        '''
18        self.literals = set(literals)
19
20
21 def __str__(self):
22     '''
23     Return the string of the clause
24     '''
25     return "[" + ' '.join([str(l)
26     ↪ for l in self.literals]) + "]"
27
28 def __eq__(self, other):
29     '''
30     Check if two clauses are the same
31     '''
32     return self.literals == other.literals

```

```

1 class KB:
2     '''
3     A knowledge base is a set of clauses
4     '''
5     def __init__(self, clauses=set()):
6         '''
7         Initialize the knowledge base
8
9         Parameters
10        -----
11        clauses : set
12            The set of clauses
13
14        Returns
15        -----
16        None
17        '''
18        self.clauses = clauses
19
20 def insert(self, clause: Clause, KB0):
21     '''
22     Insert a clause into the knowledge base
23
24     Parameters
25     -----
26     clause : Clause
27         The clause to be inserted
28     KB0 : KB
29         The knowledge base that
30     ↪ contains of clauses that are already inferred

```

```

30
31 Returns
32 -----
33 None
34 '''
35 for clause1 in KB0.clauses:
36     cell_pos = list(clause1.literals)[0].cell
37     pos = list(clause1.literals)[0].posi
38     for lit in clause.literals.copy():
39         if lit.cell == cell_pos and lit.posi !=
            ↪ = pos and lit in clause1.literals:
40             clause.literals.remove(lit)
41 if len(clause.literals) == 0:
42     return None
43 if clause in self.clauses:
44     return None
45 for clause1 in self.clauses.copy():
46     if clause1
            ↪ .literals.issubset(clause.literals):
47         return None
48     elif clause
            ↪ .literals.issubset(clause1.literals):
49         if clause1 in self.clauses:
50             self.clauses.remove(clause1)
51 if clause in KB0.clauses or clause in self
            ↪ .clauses or len(clause.literals) == 0:
52     return None
53
54 if len(clause.literals) >= 1:
55     self.clauses.add(clause)
56     # print(f"insert {clause}")
57     # print(f"[\n{'', '.join([str(c)
            ↪ for c in self.clauses])}\n]")

```

Code 5: Player Module

```

1 class Player:
2     '''
3     The player class
4     '''
5     def __init__(self, game: Game):
6         '''
7         Initialize the player
8
9         Parameters
10        -----
11        game : Game
12            The game to be played
13
14        Returns
15        -----

```

```

16 None
17 '''
18 self.game = game
19 self.KB = KB(set())
20 self.KB0 = KB(set())
21 self.mine = set()
22 self.safe = set()
23 for i in self.game.get_init_safe_cells():
24     self.safe.add(i)
25     self.KB.insert(Clause([Literal(i,
            ↪ False])), self.KB0)
26
27 def play(self):
28     '''
29     Play the game
30
31     Parameters
32     -----
33     None
34
35     Returns
36     -----
37     None
38     '''
39     unmarked_cnt = 0
40     while unmarked_cnt <= 10:
41         self.game.print_board()
42         print(f"# in KB: {len(self.KB.clauses)},
            ↪ # in KB0: {len(self.KB0.clauses)}")
43         # for clause in self.KB.clauses:
44         #     print(clause, len(clause))
45         # print("----")
46         # for clause in self.KB0.clauses:
47         #     print(clause)
48         print(f"# single clause in
            ↪ KB: {len([clause for clause in self
            ↪ .KB.clauses if len(clause) == 1])}")
49         updated = False
50         if Clause([]) in self.KB.clauses:
51             self.KB.clauses.remove(Clause([]))
52         for clause in self.KB.clauses:
53             if len(clause) == 1:
54                 unmarked_cnt = 0
55                 updated = True
56                 lit = list(clause.literals)[0]
57                 self.KB.clauses.remove(clause)
58                 self.KB0
                    ↪ .clauses.add(Clause(clause
                    ↪ .literals.copy()))
59                 print(f"Open cell
                    ↪ {lit.cell} with {lit.posi}")
60                 if lit.posi:

```



```

61         if self.game.open_cell(lit
        ↪ .cell, False) == -1:
62             print('Game Over!')
63             exit(0)
64             self.mine.add(lit.cell)
65     else:
66         if self.game.open_cell(lit
        ↪ .cell, True) == -1:
67             print('Game Over!')
68             exit(0)
69             self.safe.add(lit.cell)
70     # for clause1
71     ↪ in self.KB.clauses.copy():
72     # print(clause1)
73     for clause1
74     ↪ in self.KB.clauses.copy():
75         if clause1 in self.KB.clauses:
76             self.KB
77             ↪ .clauses.remove(clause1)
78
79     ↪ a,
80     ↪ b
81     ↪ =
82     ↪ matching_clauses(Clause(clause
83     ↪ .literals.copy()), clause1)
84     if a:
85         self.KB.insert(a, self.KB0)
86     if b:
87         self.KB.insert(b, self.KB0)
88
89     if not lit.posi:
90         pos, n = self
91         ↪ .game.get_hint(lit.cell)
92         # print(pos, n)
93         if len(pos) == n:
94             for i in pos:
95
96                 ↪ self
97                 ↪ .KB
98                 ↪ .insert(Clause([Literal(i,
99                 ↪ True)]), self.KB0)
100     elif n == 0:
101         for i in pos:
102
103             ↪ self
104             ↪ .KB
105             ↪ .insert(Clause([Literal(i,
106             ↪ False)]), self.KB0)
107     else:
108         for comb
109         ↪ in combinations(pos,
110         ↪ len(pos)-n+1):
111             lits = []

```

```

93         for cell in comb:
94
95             ↪ lits
96             ↪ .append(Literal(cell,
97             ↪ True))
98             self.KB
99             ↪ .insert(Clause(lits),
100             ↪ self.KB0)
101         for comb in
102         ↪ combinations(pos, n+1):
103             lits = []
104             for cell in comb:
105
106                 ↪ lits
107                 ↪ .append(Literal(cell,
108                 ↪ False))
109             self.KB
110             ↪ .insert(Clause(lits),
111             ↪ self.KB0)
112
113         break
114     if updated:
115         continue
116     KB_clause = list(self.KB.clauses.copy())
117     print("entering pairwise matching")
118     unmarked_cnt += 1
119     for idx, i in tqdm(enumerate(KB_clause)):
120         for j in KB_clause[idx+1:]:
121             if i in self.KB.clauses:
122                 self.KB.clauses.remove(i)
123             if j in self.KB.clauses:
124                 self.KB.clauses.remove(j)
125             if (len(i) == 0 or len(j) == 0):
126                 continue
127             a, b = matching_clauses(Clause(i
128             ↪ .literals.copy()),
129             ↪ Clause(j.literals.copy()))
130
131             if a:
132                 self.KB.insert(a, self.KB0)
133                 if a != i and a != j:
134                     # print(len(i),
135                     ↪ len(j), i, j, a, b)
136                     updated = True
137             else:
138                 # print(len(i),
139                 ↪ len(j), i, j, a, b)
140                 updated = True
141             if b:
142                 self.KB.insert(b, self.KB0)
143                 if b != j and b != i:
144                     # print(len(i),
145                     ↪ len(j), i, j, a, b)
146                     updated = True

```

```

130         else:
131             # print(len(i),
132             ↪ len(j), i, j, a, b)
133             updated = True
134
135         if not updated:
136             if len(self.KB0.clauses)
137             ↪ != self.game.h * self.game.w:
138                 print("Stuck")
139             else:
140                 print("Win!")
141                 exit(0)
142     print("Stuck")
143     exit(0)

```

Code 6: matching_clauses Function

```

34         for j in b.literals:
35             if i.cell == j.cell and i.posi != j.posi:
36                 comps.add(i.cell)
37
38         if len(comps) == 1:
39             a = Clause(list(a.literals
40             ↪ .copy().union(b.literals.copy())))
41             b = None
42             for i in comps:
43                 if Literal(i, True) in a.literals:
44                     a.literals.remove(Literal(i, True))
45                 if Literal(i, False) in a.literals:
46                     a.literals.remove(Literal(i, False))
47             return a, None
48
49     return (a if len(a.literals) > 0 else
50     ↪ None), (b if len(b.literals) > 0 else None)

```

Code 7: Main function

```

1 def matching_clauses(a: Clause, b: Clause):
2     '''
3     Check
4     ↪ if two clauses can be matched using resolution
5
6     Parameters
7     -----
8     a : Clause
9         The first clause
10    b : Clause
11        The second clause
12
13    Returns
14    -----
15    a : Clause
16        The first clause after matching
17    b : Clause
18        The second clause after matching
19    '''
20    if a == b:
21        return a, None
22
23    if a.literals.issubset(b.literals):
24        return a, None
25
26    if b.literals.issubset(a.literals):
27        return b, None
28
29    if len(a) > 2 and len(b) > 2:
30        return a, b
31
32    comps = set()
33    for i in a.literals:

```

```

1 if __name__ == "__main__":
2     parser = argparse.ArgumentParser()
3     parser.add_argument(
4         "-d",
5         "--difficulty",
6         dest="difficulty",
7         type=int,
8         help="difficulty level
9         ↪ of the game 0: easy, 1: medium, 2: hard",
10        default=0
11    )
12    parser.add_argument(
13        "-n",
14        dest="n",
15        type=int,
16        help="number of games to play",
17        default=1
18    )
19    args = parser.parse_args()
20    records = [] # win, time
21    for i in range(args.n):
22        game = Game(args.difficulty)
23        start = time.time()
24        player = Player(game)
25        res = player.play()
26        end = time.time()
27        records.append((res, end-start))
28    # Print status with windows
29
30    print(f"Play {args.n}
31    ↪ games with difficulty {args.difficulty}")

```

```

30     print(f"Win:
    ↳ {len([i for i in records if i[0] == 1])}")
31     print(f"Lose:
    ↳ {len([i for i in records if i[0] == -1])}")
32     print(f"Stuck:
    ↳ {len([i for i in records if i[0] == 0])}")
33     print(f"Average time: {round(sum([i[1]
    ↳ for i in records])/len(records), 2)} sec.")

```

Appendix B Complete Code

One can run this Python code by `python main.py [- d difficulty]`. Can also find the code from <https://github.com/jayin92/NYCU-AI-Capstone/blob/main/hw3/src/main.py>.