# Artificial Intelligence Capstone Project3 report

Author: 109550093 黃得誠

## I. Introduction

In this project, I solve the minesweeper game using logical agents. In the minesweeper game, every cell has only two kinds of value, true for mine and false for safe. The logical agent will make moves base on its knowledge base (KB). During the game, the logical agent will update its KB, by adding new information or adapt to environment changes. In this project, the game will be initialized by three level, easy, medium, or hard, which indicates the number of cells and mines. An initial list of "safe" cells is also created, the number of initial safe cells is a factor that I can modify. The three levels are, Easy (9x9 board with 10 mines), Medium (16x16 board with 25 mines), and Hard (30x16 board with 99 mines). I start with round(sqrt(#cells)) for number of initial safe cells.

## II. Environment

I implemented this logical minesweeper agent in Python.

## III. Algorithms and implementation

For implementation, I created three classes, "MineSweeper" as the game control module, "Board" as the game board information, and "PlayerAgent" as player module.

```
15  class MineSweeper:
16      def __init__(self, level, free):
17          self.board = Board(level, free)
18          # print(initial_board.ans, initial_board.init_safe_cell)
19          self.playerAgent = PlayerAgent(self.board)
20          # print(playerAgent.KB, playerAgent.KB0)
21          # print(playerAgent.board.board_size)
22          # self.foldername = f'{level}_{self.board.init_safe_cell_num}_{int(time.time())}'
23          # os.mkdir(self.foldername)
```

At the beginning of the game, the MineSweeper initialize the game board and the PlayerAgent. When initializing the board, I generate a board with the size according to the level. I also create the initial safe cells in this section.

```
27  # GameAgent
28  class Board:
29      def __init__(self, level="easy", free=1):
30          '''
31          initialize the board information with the given level
32          '''
33          meta = board_meta[level]
34          self.board_size = meta["board_size"]
35          self.mine_num = meta["mine_num"]
36          self.annot_size = meta["annot_size"]
37          self.cell_num = self.board_size[0] * self.board_size[1]
38          self.unmark_mine_num = self.mine_num
39          self.unmark_cell_num = self.cell_num
40          self.init_safe_cell_num = int(free*(np.round(np.sqrt(self.cell_num))))
41          self.ans, self.init_safe_cell = self.generate_board()
42
```

```
    def generate_board(self):
        '''
        genearte the borad with the initialized information.

        output:
            ans: board_size numpy array storing answer
            safe_call[idx]: the initialized save call positions
        '''
```

When initializing the PlayerAgent, I initial KB with negative single-literal clauses representing all the safe cells, KB0, which is empty, status list to update the status the map.

```
class PlayerAgent:
    def __init__(self, board):
        self.board = board
        self.KB0 = {}
        self.KB = []
        init_safe_cell = self.board.init_safe_cell
        for cell in init_safe_cell:
            clause = ",".join(cell.astype(str))
            clause = f'not {clause}'
            self.KB.append([clause])

        # initialize the status
        # -1 for unmarked
        # OPEN for marked as safe
        # FLAG for marked as mine
        self.status = np.ones(self.board.board_size, dtype=np.int32) * -1
        self.cnt, self.not_found = 0, 0
```

Then the game starts, the MineSweeper will see if the game is end. If the game is not end, the game play proceeds in a loop. The game is ended when the KB is empty, meaning that all the cells are marked, succeed, or the game is stuck, meaning that no single-literal clause could be found and matching two clauses could not solve the problem either, failed.

```
    def start(self):
        # print("Game Started ...")
        while self.checkEnd():
            flag = self.playerAgent.iter()
            if flag > 0:
                self.board = self.playerAgent.board
                # if flag == 1:
                    # self.plot_board()
                    # plt.savefig(f'{self.playerAgent.cnt}_{int(time.time())}final.png', dpi=300, transparent=
                    # plt.clf()
            else:
                break
```

In each iteration, the player will first check if it is late game, which is about the global constraint. If the number of unmark cell is less than ten, the global constraint will add some clauses to the KB.

```
159         # Late game, when unmark_cell_num is less than 10, add global constraints.
160         unmark_cell_num = self.board.unmark_cell_num
161         unmark_mine_num = self.board.unmark_mine_num
162         if unmark_cell_num <= 10 and unmark_cell_num>=unmark_mine_num:
163             # print("Late Game ...")
164             unmark_cells = self.get_unmark_cells()
165             for clause in list(comb(unmark_cells, unmark_cell_num-unmark_mine_num+1)):
166                 clause = list(clause)
167                 if not self.hasduplicate(clause):
168                     self.subsumption(clause)
169             for clause in list(comb(unmark_cells, unmark_mine_num+1)):
170                 clause = list(clause)
171                 for i in range(len(clause)):
172                     clause[i] = f'not {clause[i]}'
173                 if not self.hasduplicate(clause):
174                     self.subsumption(clause)
```

Then it will enter the normal loop, which is implemented following the spec.

- The game play proceeds in a loop. Within each iteration:
  If there is a single-lateral clause in the KB:
    Mark that cell as safe or mined.
    Move that clause to KB0.

    Process the "matching" of that clause to all the remaining clauses in the KB.
      If new clauses are generated due to resolution, insert them into the KB.
    If this cell is safe:
      Query the game control module for the hint at that cell.
      Insert the clauses regarding its unmarked neighbors into the KB.
  Otherwise:
    Apply pairwise "matching" of the clauses in the KB.
      If new clauses are generated due to resolution, insert them into the KB.
        * For the step of pairwise matching, to keep the KB from growing too fast, only match clause pairs where one clause has only two literals.

The player will go through the KB, which starts from the clause that has less length (using sorted function). Once there is a single-lateral clause, check whether it is a safe case or mine case. The safe case is a bit complicated, since we need to add more clauses according to new environment information. In both cases, the player first eliminates the clauses containing both negative and positive single-literal clauses. Then I will mark the status list as FLAG or SAFE, in the case of mine or safe. In the safe case, the player will then query the hint from the board, see if there are any marked cells (if there are flag that are marked, minus one).

So far, the player has m, the number of unmarked neighbors, and n, the modified hint. There are three situations now, m=n, n=0, and m>n>0.

When m=n, meaning that all the unmarked cells are mine. Add positive-clauses to the KB.

```
219                 m = len(neighbors)
220                 n = hint
221                 # (m == n): insert the m single-literal positive clauses
222                 # to the knowledge base, one for each unmarked neighbor
223                 if m == n:
224                     for neighbor in neighbors:
225                         if not self.hasduplicate(neighbor):
226                             self.KB.append([neighbor])
```

When n=0, meaning that all the unmarked cells are safe. Add negative-clauses to the KB.

```
229                 elif n == 0:
230                     for neighbor in neighbors:
231                         if not self.hasduplicate(f'not {neighbor}'):
232                             self.KB.append([f'not {neighbor}'])
```

If m>n>0, we will generate CNV clauses into the KB.
Since there are n mines, if we pick n+1 from the m unmarked cells, there definitely will be one mine in these n+1 cells. Therefore, we can generate a all-positive-literal. On the other hand, there are m-n safe cells, if we pick m-n+1 from m cells, and there will always be one safe cell in these m-n+1 cells. An all-negative literal can be created.

```
235                 elif m > n:
236                     for clause in list(comb(neighbors, len(neighbors)-hint+1)):
237                         clause = list(clause)
238                         if not self.hasduplicate(clause):
239                             self.subsumption(clause)
240                     for clause in list(comb(neighbors, hint+1)):
241                         clause = list(clause)
242                         for i in range(len(clause)):
243                             clause[i] = f'not {clause[i]}'
244                         if not self.hasduplicate(clause):
245                             self.subsumption(clause)
```

After generating new clauses, before adding it into the knowledge base, we need to check if there are duplicated clauses in the KB. If there are not duplicated clauses in KB, do subsumption, check if there are stricter clauses in the KB. If passed the two tests, add the clause into KB.

If a single-literal clause is found during an iteration, after marking the regarding cell and updating the KB, the iteration is over. Otherwise, pairwise matching will be done to the KB. The player will go through all clauses in KB, to see if there are other duplicated clauses in the KB, or a subsumption could apply on the clauses. I implemented it using a 2-layer loop, going through all the clauses pairwise.
If neither a single-literal clause nor a update could be apply to KB, it means that the player is stuck. In this project, this means lost of the game, so the MineSweeper will end the game.

## IV. Experiments and results

### 1. Relationship between level

Results with number of initial safe cell equals to round(sqrt(number of cells)).

Matches per play means the average pairwise match used in game.

```
----------------------------------
level: easy  85/100
matches per play 0.82
----------------------------------
```

```
----------------------------------
level: medium  94/100
matches per play 0.26
----------------------------------
```

```
----------------------------------
level: hard  3/100
matches per play 5.18
----------------------------------
```

Notice that hard level stocks very often.

Win rate: Medium > Easy > Hard

Matches per play: Hard > Easy > Medium

This is the result that I expected, since the mine per cell for level is:

Medium 0.098 (25/256) < Easy 0.124 (10/81) < Hard 0.206 (99/480)

### 2. Relationship between free initial cell

Results with different initial safe cell. (0.5, 1, 2) * round(sqrt(#cells))

```
----------------------------------
times of free initial cells: 0.5
level: easy  32/50
matches per play 1.18
----------------------------------
```

```
----------------------------------
times of free initial cells: 1
level: easy  42/50
matches per play 0.46
----------------------------------
```

```
----------------------------------
times of free initial cells: 2
level: easy  43/50
matches per play 0.62
----------------------------------
```

This result is tested on easy level.

With more free initial cells, the higher win rate and lower matches per play.

This indicates that it's more likely to find a single-clause with higher free initial cells.

However, this result is not consistent with the medium level.

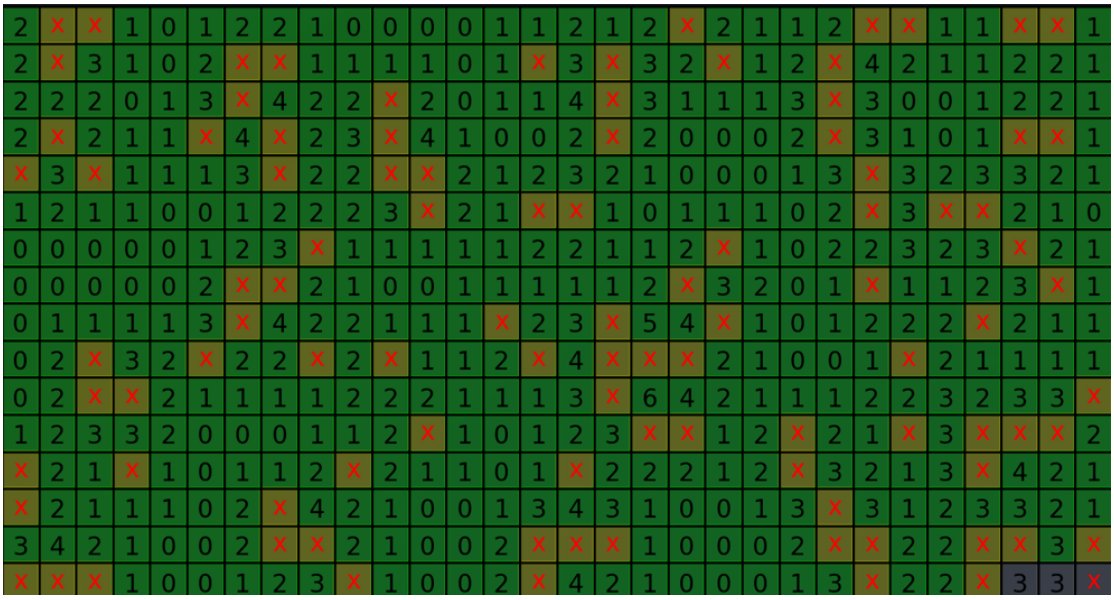
```
----------------------------------
times of free initial cells: 0.5
level: medium  48/50
matches per play 0.16
----------------------------------
----------------------------------
times of free initial cells: 1
level: medium  49/50
matches per play 0.22
----------------------------------
----------------------------------
times of free initial cells: 2
level: medium  49/50
matches per play 0.16
----------------------------------
```

1 time is better than 0.5 and 2 this time in the medium level.
Maybe that there are some relations between the free initial cell and the #mine/#cells.

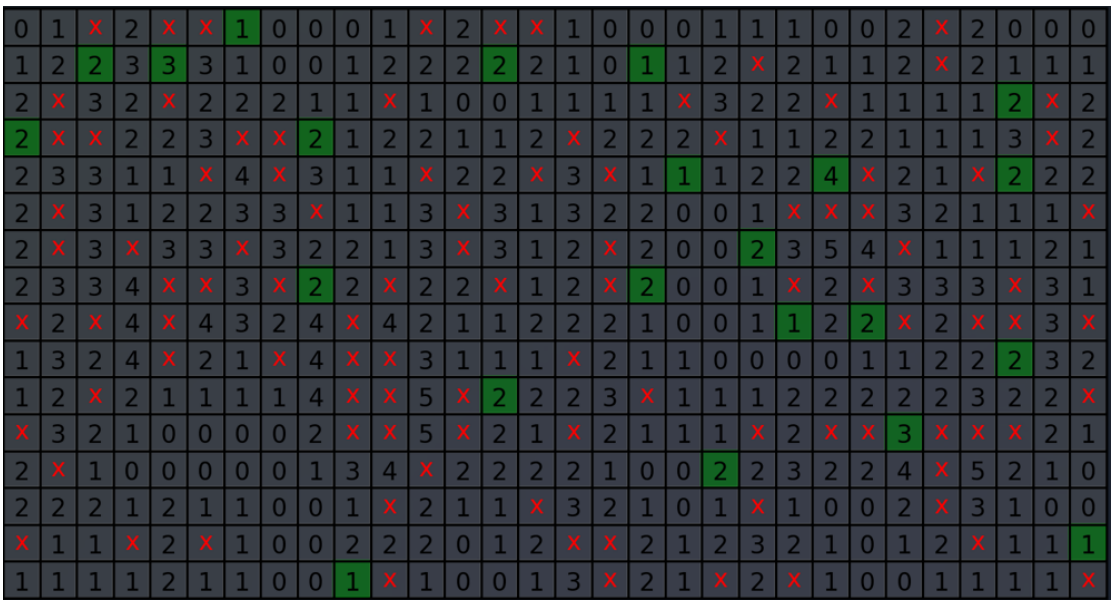
For hard situation, because the #marked_cell varies a lot, the matches per play may vary too. Therefore, I don't think it's suitable to compare it here.

### 3. Low win rate of hard level

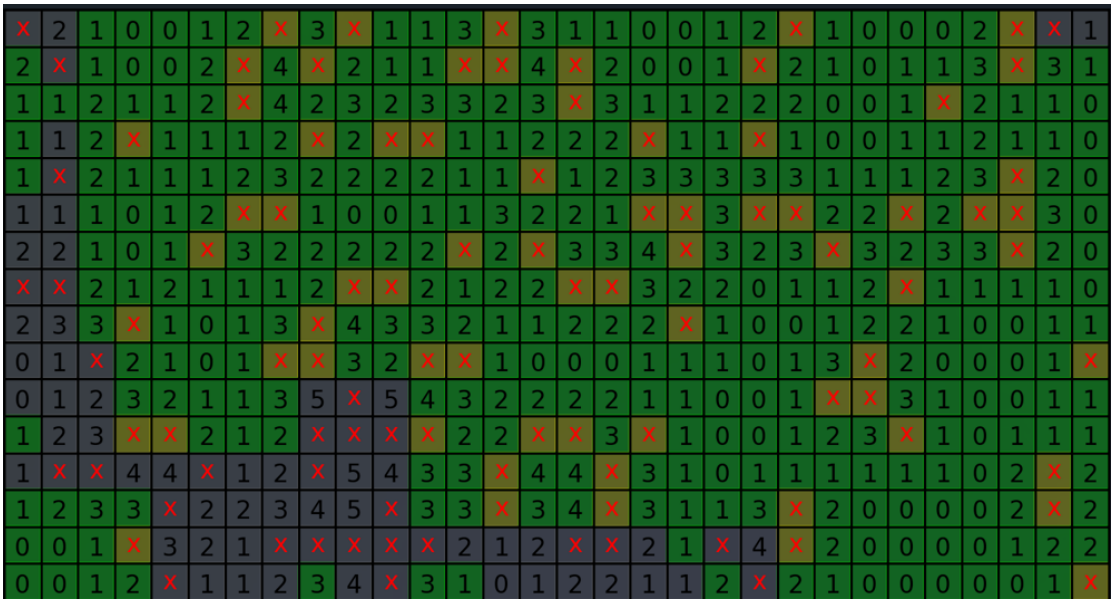
I suppose it is reasonable that hard level has a low win rate.
There are some close games where the board really can't solve using logical inference.



There are some games that we couldn’t even start, for example, if none of the free initial cells connects to each other nor a hint equal to 0 been selected as a free initial cell.



Even if the game starts successfully, it might stock during midgame also.



However, since I often play some minesweeper games, I could realize some safe steps sometimes. For example in the middle-left of the previous example,


Since the 3 has 1 mine left in its neighbor, the 2 also has 1 mine left in its neighbor, so the V marked 5 is safe in this situation.

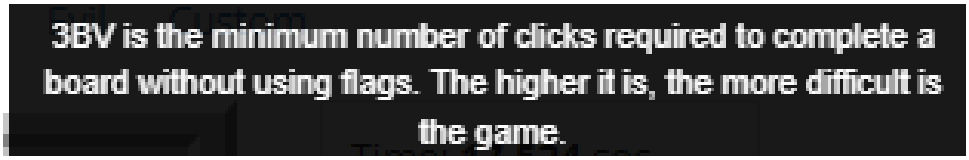

Moreover, since the left 5 is safe, the 3 in the left can infer to the bottom-right cell is mine. Maybe the game wouldn't be stuck.
I'm not sure if I implemented the algorithm wrong, or our algorithm can't solve this kind of problem.

Otherwise:
Apply pairwise "matching" of the clauses in the KB.
If new clauses are generated due to resolution, insert them into the KB.
* For the step of pairwise matching, to keep the KB from growing too fast, only match clause pairs where one clause has only two literals.

Finally, I've thought of that maybe it's because we only generate new clauses in pairwise matching with 2 literals. This may lead to the problem. However, there is a chance that I implemented the algorithm wrong.

## V. Conclusion

In this project, I've learnt the method to implement a logical agent, using CNF knowledge base. In addition, I got a chance to think of a game that I've played since I was young, minesweeper. Since I play minesweeper on minesweeper.online, I know a way to evaluate the hardness of a board, 3BV.




Maybe, it is better to find a way to evaluate the 3BV and see if there are relationships with #free initial cell.

## VI. Appendix

Main function: Main.py

HW3.py > ...

```python
import numpy as np
import matplotlib.pyplot as plt
# from itertools import combinations as comb
from collections import OrderedDict
from Board import Board
from Player import PlayerAgent
import os
import time

# Constants
MINE = 1000
OPEN = 2000
FLAG = 3000

class MineSweeper:
    def __init__(self, level, free):
        self.board = Board(level, free)
        # print(initial_board.ans, initial_board.init_safe_cell)
        self.playerAgent = PlayerAgent(self.board)
        # print(playerAgent.KB, playerAgent.KB0)
        # print(playerAgent.board.board_size)
        # self.foldername = f'{level}_{self.board.init_safe_cell_num}_{int(time.time())}'
        # os.mkdir(self.foldername)

```

```python
    def start(self):
        # print("Game Started ...")
        while self.checkEnd():
            flag = self.playerAgent.iter()
            if flag > 0:
                self.board = self.playerAgent.board
                # if flag == 1:
                    # self.plot_board()
                    # plt.savefig(f'{self.playerAgent.cnt}_{int(time.time())}final.png', dpi=300, transparent=True)
                    # plt.clf()
            else:
                break
        # print('--------------------')
        # print(f'{self.playerAgent.cnt} / {self.board.cell_num} cells marked/ all cells')
        # print(f'Pairwise matching: {self.playerAgent.not_found} times')
        # print("--------------------")
        if self.playerAgent.cnt < self.board.cell_num:
            self.plot_board()
            plt.savefig(f'test/{level}_{self.playerAgent.cnt}_{int(time.time())}final.png', dpi=300, transparent=True)
            plt.clf()
            return 0, self.playerAgent.not_found
        return 1, self.playerAgent.not_found
        # os.rename(folder, f'{folder} {cnt} {not_found}')

    def checkEnd(self):
        # print(len(self.playerAgent.KB0))
        return self.playerAgent.KB

    def plot_board(self):
        status = self.playerAgent.status
        ans = self.board.ans
        annot_size = self.board.annot_size
        l, w = status.shape
        args_mine = OrderedDict(
            color='r',
            fontsize=annot_size,
```

```python
            horizontalalignment='center',
            verticalalignment='center')
        args_hint = OrderedDict(
            fontsize=annot_size,
            horizontalalignment='center',
            verticalalignment='center')

        for i in range(l):
            for j in range(w):
                if status[i, j] == -1:
                    plt.fill([j, j, j+1, j+1], [l-i-1, l-i, l-i, l-i-1], c='gray', alpha=0.3)
                elif status[i, j] == FLAG:
                    plt.fill([j, j, j+1, j+1], [l-i-1, l-i, l-i, l-i-1], c='yellow', alpha=0.3)
                elif status[i, j] == OPEN:
                    plt.fill([j, j, j+1, j+1], [l-i-1, l-i, l-i, l-i-1], c='lime', alpha=0.3)
                if ans[i, j] == MINE:
                    plt.annotate('x', (j+0.5, l-i-0.5), **args_mine)
                else:
                    plt.annotate(ans[i, j], (j+0.5, l-i-0.6), **args_hint)

        plt.gca().set_xticks(np.arange(w+1))
        plt.gca().set_yticks(np.arange(l+1))
        plt.gca().set_xlim([0, w])
        plt.gca().set_ylim([0, l])
        plt.gca().tick_params(bottom=False, top=False, left=False, right=False)
        plt.gca().tick_params(labelbottom=False, labeltop=False, labelleft=False, labelright=False)
        plt.gca().grid(which='major', color='k', linestyle='-')
        plt.gca().set_aspect('equal')
```

```python
if __name__ == "__main__":
    level = "hard"
    timestoplay = 50
    win = 0
    num_matches = []
    frees = [0.5, 1, 2]
    for free in frees:
        for i in range(1, timestoplay+1):
            game = MineSweeper(level, free)
            flag, num = game.start()
            num_matches.append(num)
            if flag:
                win+=1
        print("---------------------------------")
        print(f"times of free initial cells: {free}")
        print(f"level: {level}  {win}/{i}")
        print(f"matches per play {sum(num_matches)/len(num_matches)}")
        print("---------------------------------")
        win = 0
        num_matches = []
```

Board.py with class of Board

```python
import numpy as np

# Constants
MINE = 1000
OPEN = 2000
FLAG = 3000

# board metadata
board_meta = {
    'easy': {
        'board_size': (9, 9),
        'mine_num': 10,
        'annot_size': 20
    },
    'medium': {
        'board_size': (16, 16),
        'mine_num': 25,
        'annot_size': 10
    },
    'hard': {
        'board_size': (16, 30),
        'mine_num': 99,
        'annot_size': 8
    }
}

```

```python
# Board
class Board:
    def __init__(self, level="easy", free=1):
        '''
        initialize the board information with the given level
        '''
        meta = board_meta[level]
        self.board_size = meta["board_size"]
        self.mine_num = meta["mine_num"]
        self.annot_size = meta["annot_size"]
        self.cell_num = self.board_size[0] * self.board_size[1]
        self.unmark_mine_num = self.mine_num
        self.unmark_cell_num = self.cell_num
        self.init_safe_cell_num = int(free*(np.round(np.sqrt(self.cell_num))))
        self.ans, self.init_safe_cell = self.generate_board()

    def generate_board(self):
        '''
        genearte the borad with the initialized information.

        output:
            ans: board_size numpy array storing answer
            safe_call[idx]: the initialized save call positions
        '''
        mines = set()
        while len(mines) < self.mine_num:
            i = np.random.randint(0, self.board_size[0])
            j = np.random.randint(0, self.board_size[1])
            mines.add((i, j))

        # Set the size of and to (board_size.x+2, board_size.y+2) for easier calculation
        ans = np.zeros((self.board_size[0]+2, self.board_size[1]+2), dtype=np.int32)
        for mine in mines:
            i, j = mine[0]+1, mine[1]+1
            ans[i, j] = MINE
            for di in [-1, 0, 1]:
                for dj in [-1, 0, 1]:
                    new_i = i + di
                    new_j = j + dj
                    if ans[new_i, new_j] == MINE:
                        continue
                    ans[new_i, new_j] += 1
        # Set ans back to (board_size.x, board_size.y)
        ans = ans[1:-1, 1:-1]
        safe_cell = np.argwhere(ans!=MINE)
        idx = np.random.choice(np.arange(0, safe_cell.shape[0]), self.init_safe_cell_num, replace=False)

        return ans, safe_cell[idx]
```

Player.py, information of class player.

```python
import numpy as np
import os
import matplotlib.pyplot as plt
from itertools import combinations as comb
# from Board import Board

# Constants
MINE = 1000
OPEN = 2000
FLAG = 3000

# PlayerAgent
class PlayerAgent:
    def __init__(self, board):
        self.board = board
        self.KB0 = {}
        self.KB = []
        init_safe_cell = self.board.init_safe_cell
        for cell in init_safe_cell:
            clause = ",".join(cell.astype(str))
            clause = f'not {clause}'
            self.KB.append([clause])

        # initialize the status
        # -1 for unmarked
        # OPEN for marked as safe
        # FLAG for marked as mine
        self.status = np.ones(self.board.board_size, dtype=np.int32) * -1
        self.cnt, self.not_found = 0, 0

    # get the eight neighbors of given cell
    def get_neighbors(self, i, j):
        neighbors = []
        for di in [-1, 0, 1]:
            for dj in [-1, 0, 1]:
                if di==0 and dj==0:
                    continue
                new_i = i + di
                new_j = j + dj
                if (new_i<0 or new_i>=self.board.board_size[0] or
                        new_j<0 or new_j>=self.board.board_size[1]):
                    continue
                neighbors.append(f'{new_i},{new_j}')
        return neighbors


    # get all unmarked cells
    def get_unmark_cells(self):
        cells = []
        for i in range(self.board.board_size[0]):
            for j in range(self.board.board_size[1]):
                if f'{i},{j}' not in self.KB0:
                    cells.append(f'{i},{j}')
        return cells

    # check whether the two clauses are identical
    # Both clause1 and clause2 are lists.
    def isduplicate_pairwise(self,clause1, clause2):
        match = 0
        for literal in clause1:
            if literal in clause2:
                match += 1
        if match==len(clause2) and len(clause1)==len(clause2):
            return True
        return False
```

```
    # check whether there exists any clause identical to the given clause in the
    # knowledge base
    # targetclause is a list contain strings in it.
    def hasduplicate(self,targetclause):
        if type(targetclause)==type(""):
            targetclause = [targetclause]
        for clause in self.KB:
            if self.isduplicate_pairwise(targetclause, clause):
                return True
        return False


    # subsumption for single-literal clause
    # clause is just a string in this function
    def literal_subsumption(self, clause):
        for i in range(len(self.KB)):
            if clause in self.KB[i]:
                self.KB[i] = []


    # subsumption between two clauses
    def subsumption_pairwise(self, i, j):
        '''
        See if there are stricter clauses that we can eliminate.
        return true if updated, false if not updated
        '''
        clause1, clause2 = self.KB[i], self.KB[j]
        match = 0
        for literal in clause1:
            if literal in clause2:
                match += 1
        if match==len(clause1) and len(clause1)<len(clause2):
            self.KB[j] = []
            return True
        elif match==len(clause2) and len(clause2)<len(clause1):
            self.KB[i] = []
            return True
        return False


    # subsumption for given clause to the knowledge base
    # clause is a list with strings in it.
    def subsumption(self, clause):
        insert = True
        update = False
        for i in range(len(self.KB)):
            sentence = self.KB[i]
            if not len(sentence):
                continue
            match = 0
            for literal in clause:
                if literal in sentence:
                    match += 1
            if match==len(sentence) and len(sentence)<=len(clause):
                insert = False
            elif match==len(clause) and len(sentence)>len(clause):
                self.KB[i] = []
                update = True
        if insert:
            self.KB.append(clause)
            update = True
        return update
```

```python
    # generate new clause if there is only one pair on complementary literals
    # between two clauses
    # both clauses are lists.
    def comp(self, clause1, clause2):
        c1 = list(clause1)
        c2 = list(clause2)
        match = 0
        for literal in clause1:
            if 'not' in literal:
                l = literal[4:]
            else:
                l = f'not {literal}'
            if l in clause2:
                c1.remove(literal)
                c2.remove(l)
                match += 1
        '''
        If there is only one pair of complementary literals:
        Apply resolution to generate a new clause, which will be inserted into the KB
        '''
        if match == 1:
            return list(set(c1+c2))
        else:
            return []
```

```python
    def iter(self):
        flag = 0
        # Late game, when unmark_cell_num is less than 10, add global constraints.
        unmark_cell_num = self.board.unmark_cell_num
        unmark_mine_num = self.board.unmark_mine_num
        if unmark_cell_num <= 10 and unmark_cell_num>=unmark_mine_num:
            # print("Late Game ...")
            unmark_cells = self.get_unmark_cells()
            for clause in list(comb(unmark_cells, unmark_cell_num-unmark_mine_num+1)):
                clause = list(clause)
                if not self.hasduplicate(clause):
                    self.subsumption(clause)
            for clause in list(comb(unmark_cells, unmark_mine_num+1)):
                clause = list(clause)
                for i in range(len(clause)):
                    clause[i] = f'not {clause[i]}'
                if not self.hasduplicate(clause):
                    self.subsumption(clause)

        # Normal Loop
        self.KB = sorted(self.KB, key=lambda c:len(c))
        found = False
        for i in range(len(self.KB)):
            clause = self.KB[i]
            # First case: If there is a single-lateral clause in KB:
            if len(clause) == 1:
                self.cnt+=1
                found = True
                # print("Step count:", self.cnt,"Clause name:", clause[0])

                # Mark that cell as safe or mined.
                # safe case
                if clause[0].startswith('not'):
                    targetname = clause[0][4:]
                    # Move the clause to KB0
                    self.KB0[targetname] = False
```

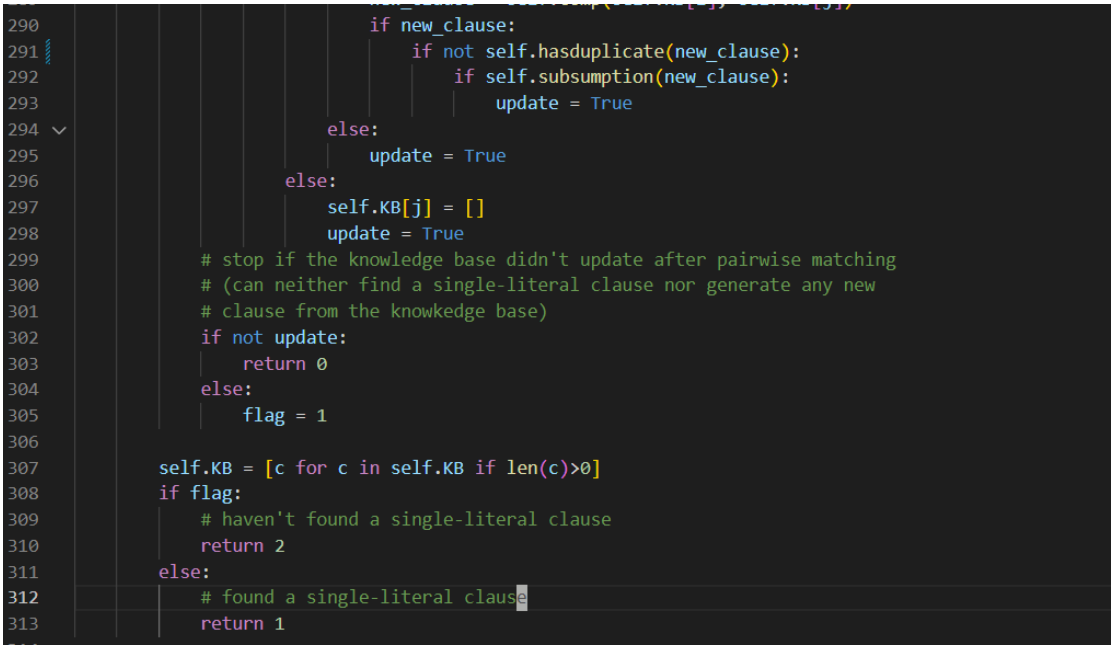
```
192                 self.KB0[targetname] = False
193                 self.KB[i] = []
194                 self.board.unmark_cell_num -=1
195 
196                 # remove clause[0] (not {targetgame})
197                 self.literal_subsumption(clause[0])
198 
199                 # Process the "matching" of that clause to all the remaining clauses in the KB.
200 
201                 # remove targetname
202                 for j in range(len(self.KB)):
203                     if targetname in self.KB[j]:
204                         self.KB[j].remove(targetname)
205                 cell = targetname.split(',')
206                 x, y = int(cell[0]), int(cell[1])
207                 self.status[x, y] = OPEN
208                 hint = self.board.ans[x, y]
209                 neighbors = self.get_neighbors(x, y)
210                 for j in range(len(neighbors)):
211                     # only consider the unmarked cells
212                     if neighbors[j] in self.KB0:
213                         if self.KB0[neighbors[j]] == True:
214                             hint -= 1
215                         neighbors[j] = ''
216                 neighbors = [n for n in neighbors if len(n) > 0]
217                 # m = number of unmarked neighbors
218                 # n = hint
219                 m = len(neighbors)
220                 n = hint
221                 # (m == n): insert the m single-literal positive clauses
222                 # to the knowledge base, one for each unmarked neighbor
223                 if m == n:
224                     for neighbor in neighbors:
225                         if not self.hasduplicate(neighbor):
```

```
                            if not self.hasduplicate(neighbor):
                                self.KB.append([neighbor])
                    # (n == 0): insert the m single-literal negative clauses
                    # to the knowledge base, one for each unmarked neighbor
                    elif n == 0:
                        for neighbor in neighbors:
                            if not self.hasduplicate(f'not {neighbor}'):
                                self.KB.append([f'not {neighbor}'])
                    # (m > n > 0): generate CNF clauses and add them to the
                    # knowledge base
                    elif m > n:
                        for clause in list(comb(neighbors, len(neighbors)-hint+1)):
                            clause = list(clause)
                            if not self.hasduplicate(clause):
                                self.subsumption(clause)
                        for clause in list(comb(neighbors, hint+1)):
                            clause = list(clause)
                            for i in range(len(clause)):
                                clause[i] = f'not {clause[i]}'
                            if not self.hasduplicate(clause):
                                self.subsumption(clause)
                    else:
                        print(f'ERROR: hint: {hint}, neighbors: {len(neighbors)}')

                # Mine case
                else:
                    # put the marked cell into KB0
                    self.KB0[clause[0]] = True
                    # remove that literal from knowledge base
                    self.KB[i]  = []
                    self.board.unmark_cell_num -= 1
                    self.board.unmark_mine_num -= 1
                    self.literal_subsumption(clause[0])
                    # clause[0] is true, so remove all not {clause[0]}
                    for j in range(len(self.KB)):
                        if f'not {clause[0]}' in self.KB[j]:
                            self.KB[j].remove(f'not {clause[0]}')
                    cell = clause[0].split(',')
                    x, y = int(cell[0]), int(cell[1])
                    self.status[x, y] = FLAG

                break

        # The case we could not find single-lateral clause in the KB.
        if not found:
            # print('Pairwise matching ...')
            update = False
            self.not_found += 1
            # print(f'size of KB: {len(self.KB)}')
            for i in range(len(self.KB)):
                if len(self.KB[i]) ==0:
                    continue
                for j in range(i+1, len(self.KB)):
                    if len(self.KB[j])==0:
                        continue
                    # check whether the two clauses are identical
                    if not self.isduplicate_pairwise(self.KB[i], self.KB[j]):
                        if not self.subsumption_pairwise(i, j):
                            '''
                            For the step of pairwise matching, to keep the KB from growing too fast, only
                            match clause pairs where one clause has only two literals.
                            '''
                            if len(self.KB[i])>2 and len(self.KB[j])>2:
                                continue
                            new_clause = self.comp(self.KB[i], self.KB[j])
```


290 if new_clause:
291 if not self.hasduplicate(new_clause):
292 if self.subsumption(new_clause):
293 update = True
294 else:
295 update = True
296 else:
297 self.KB[j] = []
298 update = True
299 # stop if the knowledge base didn't update after pairwise matching
300 # (can neither find a single-literal clause nor generate any new
301 # clause from the knowkedge base)
302 if not update:
303 return 0
304 else:
305 flag = 1
306
307 self.KB = [c for c in self.KB if len(c)>0]
308 if flag:
309 # haven't found a single-literal clause
310 return 2
311 else:
312 # found a single-literal clause
313 return 1