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 ”¬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 $x_1, x_2, \ldots, x_m$ 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 $i = 1, 2, \ldots, m$ to `KB`.
- If $n = 0$, then all the unmarked cells are safe. Therefore inserted $\neg(x_i)$ where $i = 1, 2, \ldots, m$ 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.

Table 2 shows the performance with different difficulty. For easy and medium, the game AI can solve it within 5 seconds. However, for the hard level, the execution time becomes 168.49 ± 104.7, which grows much compared with easy and medium level. This is because the process of pairwise matching and resolution grows exponentially in time.

Table 2: The execution time, win and stuck status with different difficulty. I also record the average remaining unmarked cell when stuck. For the hard level, this number is 20.33, which is not much.

Difficulty	Win	Stuck	Avg. Time	Std. Time
Easy	5	0	0.29	0.15
Medium	5	0	2.65	1.86
Hard	2	3	168.49	104.7

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.

As we can see in Table 3. I originally expect that as the number of initial safe cell increasing, the running time will decrease, because the more initial safe cells, the possibility of running pairwise decrease, thus make the program run faster. But the experiments turns out it's not the case. The first part of my guess is correct. Indeed, the number of pairwise matching decrease when the safe cell increases. However, the second part is not correct, I think the main reason is when the safe cell increases, the number of clauses in `KB` also increase. Thus, when moving clause from `KB` to `KB0`, the resolution process take more time than original setting. I also experiment the special case, which initial safe cells are all safe cells. In this case, because it won't enter the pairwise matching, the execution time is less than 64 and 128.

Table 3: The performance with different number of init safe cell. The parwise 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. And also, this game AI is not able to guess the cell when stuck. Therefore, I would like to make the game AI able to guess when stuck.

Appendix A Code of Modules and Functions

Code 1: `Game` Module

```python
class Game:
    def __init__(self, difficulty=0):
        '''
        Initialize the game board

        Parameters
        ----------
        difficulty : int
            0: Easy, 1: Medium, 2: Hard

        Returns
        -------
        None
        '''
        board_configurations = [
            (9, 9, 10),   # Easy
            (16, 16, 25), # Medium
            (16, 30, 99)  # Hard
        ]
        self.h, self.w, self.num_of_mines = board_configurations[difficulty] # height, width, number of mines
        self.board = [[0 for _ in range(self.w)] for _ in range(self.h)] # -1: mine, 0~8: number of mines around
        self.shown_cell = [[False for _ in range(self.w)] for _ in range(self.h)] # Indicate the cell is opened or not
        self.mine_pos = set() # The position of mines

        # Randomly generate mines
        while len(self.mine_pos) < self.num_of_mines:
            i = random.randrange(self.h)
            j = random.randrange(self.w)
            if (i, j) not in self.mine_pos:
                self.mine_pos.add((i, j))
                self.board[i][j] = -1

    def open_cell(self, cell, safe):
        '''
        Open the cell and return the number of mines around the cell

        Parameters
        ----------
        cell : tuple
            The position of the cell
        safe : bool
            True if the cell is safe, False if the cell is a mine

        Returns
        -------
        int
            The number of mines around the cell, return -1 if wrongly opened
        '''
        if ((cell in self.mine_pos) ^ (not safe)) or self.shown_cell[cell[0]][cell[1]]:
            return -1
        if cell not in self.mine_pos:
            self.board[cell[0]][cell[1]] = self.get_surround_mines(cell)
        else:
            self.board[cell[0]][cell[1]] = "X"

        self.shown_cell[cell[0]][cell[1]] = True

        return self.board[cell[0]][cell[1]]

    def get_hint(self, cell):
        '''
        Get the hint of the cell

        Parameters
        ----------
        cell : tuple
            The position of the cell

        Returns
        -------
        list
            The list of the cells around the cell
        int
            The number of mines around the cell
        '''
        cnt = 0
        res = []
        for i in range(cell[0]-1, cell[0]+2):
            for j in range(cell[1]-1, cell[1]+2):
                if i < 0 or i >= self.h or j < 0 or j >= self.w:
                    continue
                if self.shown_cell[i][j]:
                    continue
                if (i, j) != cell:
                    if (i, j) in self.mine_pos:
                        cnt += 1
                    res.append((i, j))
```

```python
        return res, cnt

    def get_surround_mines(self, cell):
        '''
        Get the number of mines around the cell

        Parameters
        ----------
        cell : tuple
            The position of the cell

        Returns
        -------
        int
            The number of mines around the cell
        '''
        cnt = 0
        for i in range(cell[0]-1, cell[0]+2):
            for j in range(cell[1]-1, cell[1]+2):
                if (i, j) in self.mine_pos:
                    cnt += 1
        return cnt


    def get_init_safe_cells(self):
        '''
        Get the initial safe cells

        Parameters
        ----------
        None

        Returns
        -------
        set
            The set of the initial safe cells
        '''
        num = round(math.sqrt(self.h * self.w))
        # num = 10
        init_cells = set()
        while len(init_cells) < num:
            i = random.randrange(self.h)
            j = random.randrange(self.w)
            if (i, j) not in self.mine_pos and (i, j) not in init_cells:
                init_cells.add((i, j))

        return init_cells

    def print_board(self):
        '''
        Print the game board. ? means the cell is not opened yet. X means the cell is a mine. 0~8 means the number of mines around the cell.

        Parameters
        ----------
        None

        Returns
        -------
        None
        '''
        os.system('cls')
        for i in range(self.h):
            for j in range(self.w):
                if self.shown_cell[i][j]:
                    print(self.board[i][j], end=' ')
                else:
                    print('?', end=' ')
            print()
```

Code 2: `Literal` Module

```python
class Literal:
    '''
    A literal is a cell with a positive or negative sign. For example, (0, 0) is a positive literal, and (0, 0)' is a negative literal.
    '''
    def __init__(self, cell, is_posi):
        '''
        Initialize the literal

        Parameters
        ----------
        cell : tuple
            The position of the cell
        is_posi : bool
            True if the literal is positive, False if the literal is negative

        Returns
        -------
        None
        '''
        self.cell = cell
        self.posi = is_posi

    def __eq__(self, other):
        '''
        Check if two literals are the same
```

```
        '''
        return self.cell == other.cell and self.posi == other.posi

    def __str__(self):
        '''
        Return the string of the literal
        '''
        return str(self.cell) + ('' if self.posi else ”'”)

    def __hash__(self):
        '''
        Return the hash value of the literal
        '''
        return hash(str(self))
```

Code 3: `Clause` Module

```
class Clause:
    '''
    A clause is a set of literals
    '''
    def __init__(self, literals=[]):
        '''
        Initialize the clause

        Parameters
        ----------
        literals : list
            The list of literals

        Returns
        -------
        None
        '''
        self.literals = set(literals)


    def __str__(self):
        '''
        Return the string of the clause
        '''
        return ”[” + ' '.join([str(l) for l in self.literals]) + ”]”

    def __eq__(self, other):
        '''
        Check if two clauses are the same
        '''
        return self.literals == other.literals
```

```

    def __len__(self):
        '''
        Return the number of literals in the clause
        '''
        return len(self.literals)

    def __hash__(self):
        '''
        Return the hash value of the clause
        '''
        return hash(str(self))

    def __copy__(self):
        '''
        Return the copy of the clause
        '''
        return Clause(self.literals.copy())
```

Code 4: `KB` Module

```
class KB:
    '''
    A knowledge base is a set of clauses
    '''
    def __init__(self, clauses=set()):
        '''
        Initialize the knowledge base

        Parameters
        ----------
        clauses : set
            The set of clauses

        Returns
        -------
        None
        '''
        self.clauses = clauses

    def insert(self, clause: Clause, KB0):
        '''
        Insert a clause into the knowledge base

        Parameters
        ----------
        clause : Clause
            The clause to be inserted
        KB0 : KB
            The knowledge base that contains of claueses that are already inferred
```

```python

        Returns
        -------
        None
        '''
        for clause1 in KB0.clauses:
            cell_pos = list(clause1.literals)[0].cell
            pos = list(clause1.literals)[0].posi
            for lit in clause.literals.copy():
                if lit.cell == cell_pos and lit.posi != pos and lit in clause1.literals:
                    clause.literals.remove(lit)
        if len(clause.literals) == 0:
            return None
        if clause in self.clauses:
            return None
        for clause1 in self.clauses.copy():
            if clause1.literals.issubset(clause.literals):
                return None
            elif clause.literals.issubset(clause1.literals):
                if clause1 in self.clauses:
                    self.clauses.remove(clause1)
        if clause in KB0.clauses or clause in self.clauses or len(clause.literals) == 0:
            return None

        if len(clause.literals) >= 1:
            self.clauses.add(clause)
        # print(f"insert {clause}")
        # print(f"[\n{','.join([str(c) for c in self.clauses])}\n]")
```

Code 5: `Player` Module

```python
class Player:
    '''
    The player class
    '''
    def __init__(self, game: Game):
        '''
        Initialize the player

        Parameters
        ----------
        game : Game
            The game to be played

        Returns
        -------
        None
        '''
        self.game = game
        self.KB = KB(set())
        self.KB0 = KB(set())
        self.mine = set()
        self.safe = set()
        for i in self.game.get_init_safe_cells():
            self.safe.add(i)
            self.KB.insert(Clause([Literal(i, False)]), self.KB0)

    def play(self):
        '''
        Play the game

        Parameters
        ----------
        None

        Returns
        -------
        None
        '''
        unmarked_cnt = 0
        while unmarked_cnt <= 10:
            self.game.print_board()
            print(f"# in KB: {len(self.KB.clauses)}, # in KB0: {len(self.KB0.clauses)}")
            # for clause in self.KB.clauses:
            #     print(clause, len(clause))
            # print("----")
            # for clause in self.KB0.clauses:
            #     print(clause)
            print(f"# single clause in KB: {len([clause for clause in self.KB.clauses if len(clause) == 1])}")
            updated = False
            if Clause([]) in self.KB.clauses:
                self.KB.clauses.remove(Clause([]))
            for clause in self.KB.clauses:
                if len(clause) == 1:
                    unmarked_cnt = 0
                    updated = True
                    lit = list(clause.literals)[0]
                    self.KB.clauses.remove(clause)
                    self.KB0.clauses.add(Clause(clause.literals.copy()))
                    print(f"Open cell {lit.cell} with {lit.posi}")
                    if lit.posi:
```

```python
                if self.game.open_cell(lit.cell, False) == -1:
                    print('Game Over!')
                    exit(0)
                self.mine.add(lit.cell)
            else:
                if self.game.open_cell(lit.cell, True) == -1:
                    print('Game Over!')
                    exit(0)
                self.safe.add(lit.cell)
            # for clause1 in self.KB.clauses.copy():
            #     print(clause1)
            for clause1 in self.KB.clauses.copy():
                if clause1 in self.KB.clauses:
                    self.KB.clauses.remove(clause1)
                a, b = matching_clauses(Clause(clause.literals.copy()), clause1)
                if a:
                    self.KB.insert(a, self.KB0)
                if b:
                    self.KB.insert(b, self.KB0)

            if not lit.posi:
                pos, n = self.game.get_hint(lit.cell)
                # print(pos, n)
                if len(pos) == n:
                    for i in pos:
                        self.KB.insert(Clause([Literal(i, True)]), self.KB0)
                elif n == 0:
                    for i in pos:
                        self.KB.insert(Clause([Literal(i, False)]), self.KB0)
                else:
                    for comb in combinations(pos, len(pos)-n+1):
                        lits = []
                        for cell in comb:
                            lits.append(Literal(cell, True))
                        self.KB.insert(Clause(lits), self.KB0)
                    for comb in combinations(pos, n+1):
                        lits = []
                        for cell in comb:
                            lits.append(Literal(cell, False))
                        self.KB.insert(Clause(lits), self.KB0)
                break
        if updated:
            continue
        KB_clause = list(self.KB.clauses.copy())
        print(”entering pairwise matching”)
        unmarked_cnt += 1
        for idx, i in tqdm(enumerate(KB_clause)):
            for j in KB_clause[idx+1:]:
                if i in self.KB.clauses:
                    self.KB.clauses.remove(i)
                if j in self.KB.clauses:
                    self.KB.clauses.remove(j)
                if(len(i) == 0 or len(j) == 0):
                    continue
                a, b = matching_clauses(Clause(i.literals.copy()), Clause(j.literals.copy()))

                if a:
                    self.KB.insert(a, self.KB0)
                    if a != i and a != j:
                        # print(len(i), len(j), i, j, a, b)
                        updated = True
                else:
                    # print(len(i), len(j), i, j, a, b)
                    updated = True
                if b:
                    self.KB.insert(b, self.KB0)
                    if b != j and b != i:
                        # print(len(i), len(j), i, j, a, b)
                        updated = True
```

```
                else:
                    # print(len(i), len(j), i, j, a, b)
                    updated = True

            if not updated:
                if len(self.KB0.clauses) != self.game.h * self.game.w:
                    print(”Stuck”)
                else:
                    print(”Win!”)
                exit(0)
        print(”Stuck”)
        exit(0)
```

Code 6: `matching_clauses` Function

```
def matching_clauses(a: Clause, b: Clause):
    '''
    Check if two clauses can be matched using resolution

    Parameters
    ----------
    a : Clause
        The first clause
    b : Clause
        The second clause

    Returns
    -------
    a : Clause
        The first clause after matching
    b : Clause
        The second clause after matching
    '''
    if len(a) > 2 and len(b) > 2:
        return a, b

    if a == b:
        return a, None

    if a.literals.issubset(b.literals):
        return a, None

    if b.literals.issubset(a.literals):
        return b, None

    a = Clause(a.literals.copy())
    b = Clause(b.literals.copy())

    comps = set()
    for i in a.literals:
        for j in b.literals:
            if i.cell == j.cell and i.posi != j.posi:
                comps.add(i.cell)

    if len(comps) == 1:
        a = Clause(list(a.literals.copy().union(b.literals.copy())))
        b = None
        for i in comps:
            if Literal(i, True) in a.literals:
                a.literals.remove(Literal(i, True))
            if Literal(i, False) in a.literals:
                a.literals.remove(Literal(i, False))
        return a, None

    return (a if len(a.literals) > 0 else None), (b if len(b.literals) > 0 else None)
```

Code 7: Main function

```
if __name__ == ”__main__”:
    parser = argparse.ArgumentParser()
    parser.add_argument(
        ”-d”,
        ”--difficulty”,
        dest=”difficulty”,
        type=int,
        help=”difficulty level of the game 0: easy, 1: medium, 2: hard”,
        default=0
    )
    parser.add_argument(
        ”-n”,
        dest=”n”,
        type=int,
        help=”number of games to play”,
        default=1
    )
    args = parser.parse_args()
    records = [] # win, time
    for i in range(args.n):
        game = Game(args.difficulty)
        start = time.time()
        player = Player(game)
        res = player.play()
        end = time.time()
        records.append((res, end-start))
    # Print status with windows
```

```python
    print(f”Play {args.n} games with difficulty {args.difficulty}”)
    print(f”Win: {len([i for i in records if i[0] == 1])}”)
    print(f”Lose: {len([i for i in records if i[0] == -1])}”)
    print(f”Stuck: {len([i for i in records if i[0] == 0])}”)
    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`.