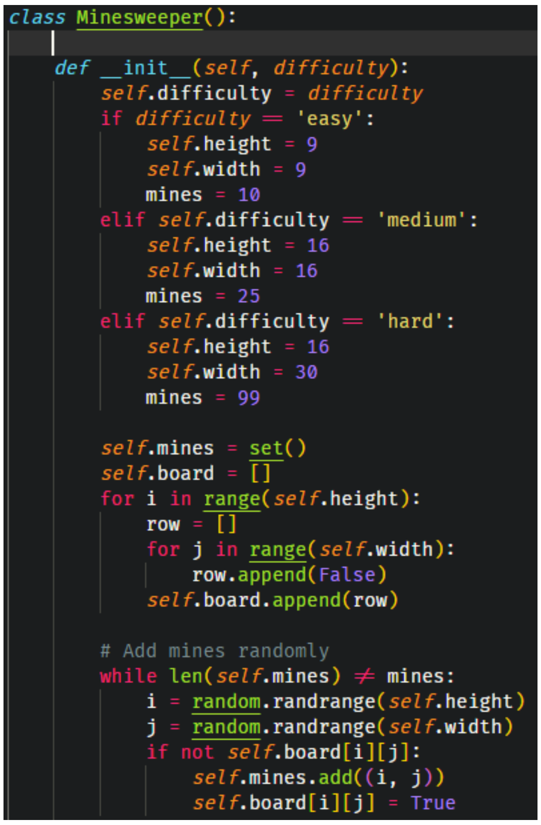
AI Prog3

109550087 單宇晟

Game control module

● Initialize

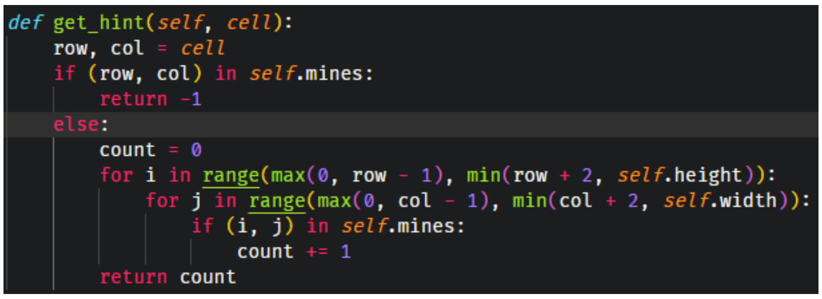
When the game starts, we can decide the difficulty of the game for AI to play. And this will initialize the board as the spec said.



```
class Minesweeper():

    def __init__(self, difficulty):
        self.difficulty = difficulty
        if difficulty == 'easy':
            self.height = 9
            self.width = 9
            mines = 10
        elif self.difficulty == 'medium':
            self.height = 16
            self.width = 16
            mines = 25
        elif self.difficulty == 'hard':
            self.height = 16
            self.width = 30
            mines = 99

        self.mines = set()
        self.board = []
        for i in range(self.height):
            row = []
            for j in range(self.width):
                row.append(False)
            self.board.append(row)

        # Add mines randomly
        while len(self.mines) != mines:
            i = random.randrange(self.height)
            j = random.randrange(self.width)
            if not self.board[i][j]:
                self.mines.add((i, j))
                self.board[i][j] = True
```

● Provide the hint

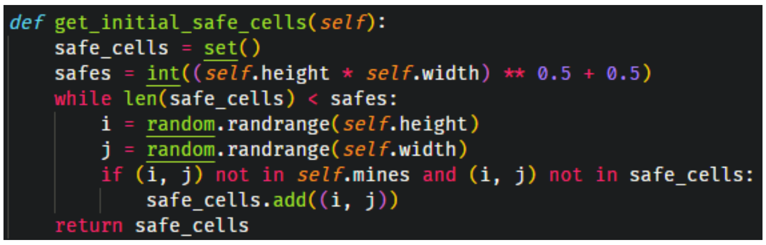
get_hint() will provide each cell the number of its neighboring mines, which help us make clauses later.



```
def get_hint(self, cell):
    row, col = cell
    if (row, col) in self.mines:
        return -1
    else:
        count = 0
        for i in range(max(0, row - 1), min(row + 2, self.height)):
            for j in range(max(0, col - 1), min(col + 2, self.width)):
                if (i, j) in self.mines:
                    count += 1
        return count
```

● Provide an initial list of "safe" cells

Since the first few clicks of a Minesweeper game are usually random clicks, to avoid losing at the first step, get_initial_safe_cells() provide a list of some safe cells to click in the early game stage. The number of initial safe cells are round(sqrt(#cells)). Also, this provide an additional information of the game board, which makes AI start to gain knowledge from those cells.

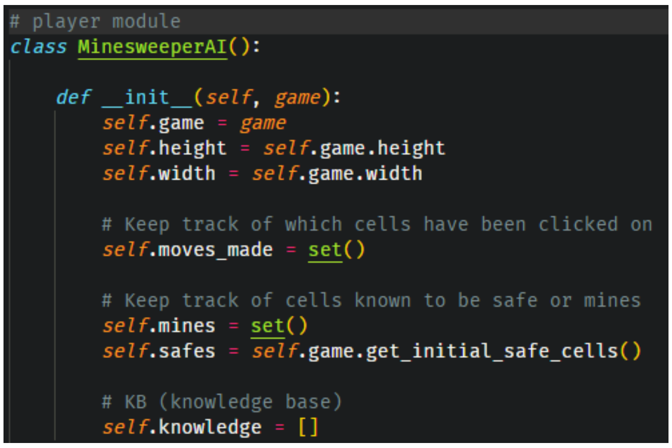


```python
def get_initial_safe_cells(self):
    safe_cells = set()
    safes = int((self.height * self.width) ** 0.5 + 0.5)
    while len(safe_cells) < safes:
        i = random.randrange(self.height)
        j = random.randrange(self.width)
        if (i, j) not in self.mines and (i, j) not in safe_cells:
            safe_cells.add((i, j))
    return safe_cells
```

Player module

- Initialize

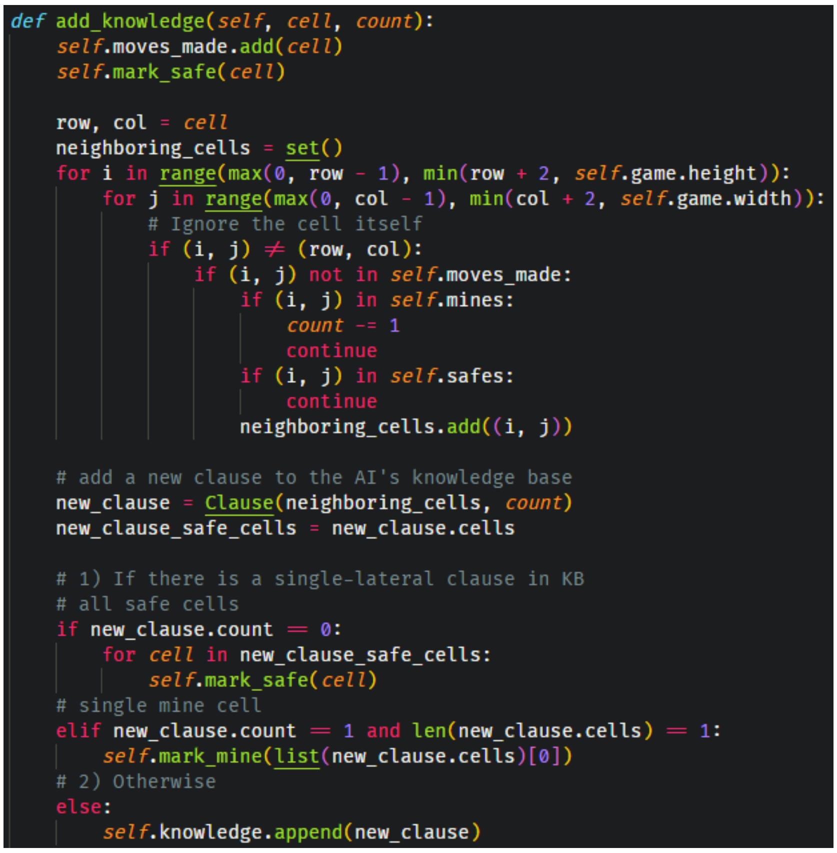
Here, self.knowledge is the knowledge base (KB) in my program, it would save clauses which were gained during playing.



```python
# player module
class MinesweeperAI():

    def __init__(self, game):
        self.game = game
        self.height = self.game.height
        self.width = self.game.width

        # Keep track of which cells have been clicked on
        self.moves_made = set()

        # Keep track of cells known to be safe or mines
        self.mines = set()
        self.safes = self.game.get_initial_safe_cells()

        # KB (knowledge base)
        self.knowledge = []
```

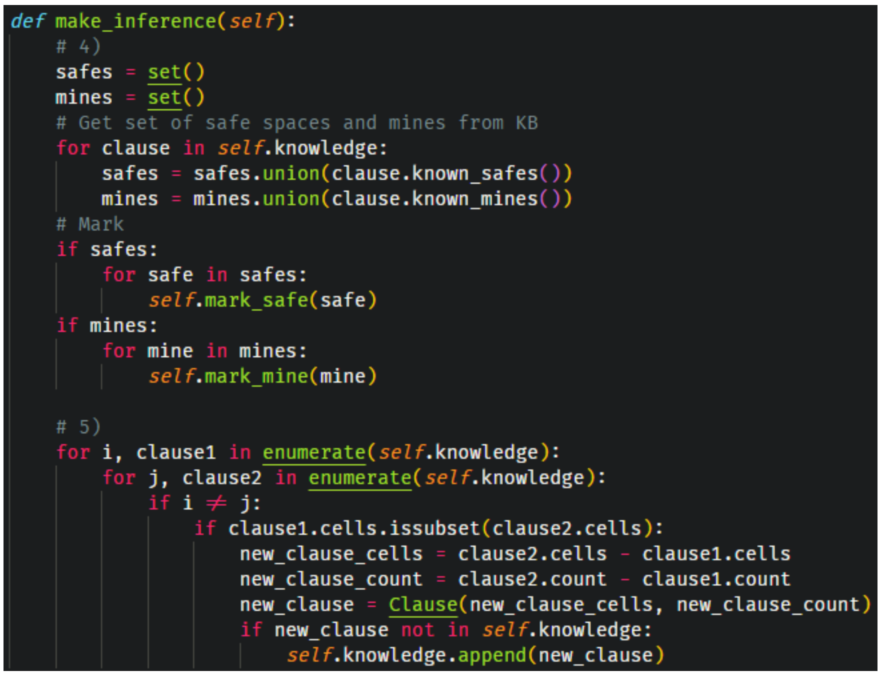
Game flow

In general, we first make a safe move (the first move will be chosen from initial safe cells) and mark it as a safe cell. Next, I find the remaining neighbors which is not a mine, and make this clue into a new clause. If the new clause is a single-lateral clause, mark it as safe or mine.



```python
def add_knowledge(self, cell, count):
    self.moves_made.add(cell)
    self.mark_safe(cell)

    row, col = cell
    neighboring_cells = set()
    for i in range(max(0, row - 1), min(row + 2, self.game.height)):
        for j in range(max(0, col - 1), min(col + 2, self.game.width)):
            # Ignore the cell itself
            if (i, j) ≠ (row, col):
                if (i, j) not in self.moves_made:
                    if (i, j) in self.mines:
                        count -= 1
                        continue
                    if (i, j) in self.safes:
                        continue
                    neighboring_cells.add((i, j))

    # add a new clause to the AI's knowledge base
    new_clause = Clause(neighboring_cells, count)
    new_clause_safe_cells = new_clause.cells

    # 1) If there is a single-lateral clause in KB
    # all safe cells
    if new_clause.count == 0:
        for cell in new_clause_safe_cells:
            self.mark_safe(cell)
    # single mine cell
    elif new_clause.count == 1 and len(new_clause.cells) == 1:
        self.mark_mine(list(new_clause.cells)[0])
    # 2) Otherwise
    else:
        self.knowledge.append(new_clause)
```

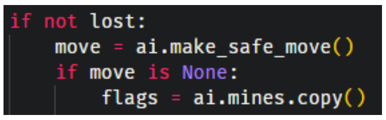
After add_knowledge(), I start to organize those clauses AI has so far, which is the “matching” part in the spec. In the beginning, I combine safe cells and mine among all clauses, mark them as safe or mine. Then I check if any two different clause has a “subset” relationship. If a clause is a subset of another, we can know that the rest of the larger clause (except the smaller clause part) can become another clause, too. Moreover, if the new clause we find hasn’t been in the knowledge base, add it into our knowledge base.



```python
def make_inference(self):
    # 4)
    safes = set()
    mines = set()
    # Get set of safe spaces and mines from KB
    for clause in self.knowledge:
        safes = safes.union(clause.known_safes())
        mines = mines.union(clause.known_mines())
    # Mark
    if safes:
        for safe in safes:
            self.mark_safe(safe)
    if mines:
        for mine in mines:
            self.mark_mine(mine)

    # 5)
    for i, clause1 in enumerate(self.knowledge):
        for j, clause2 in enumerate(self.knowledge):
            if i != j:
                if clause1.cells.issubset(clause2.cells):
                    new_clause_cells = clause2.cells - clause1.cells
                    new_clause_count = clause2.count - clause1.count
                    new_clause = Clause(new_clause_cells, new_clause_count)
                    if new_clause not in self.knowledge:
                        self.knowledge.append(new_clause)
```

Game termination

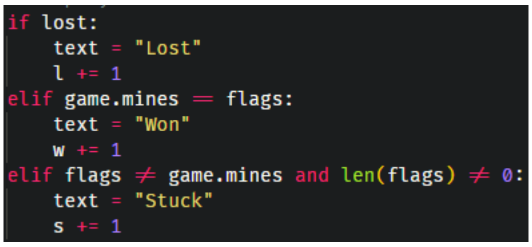
While the game state is not lost, AI will keep making moves. However, if there are no more safe moves, save the mines' location (flag), this help us decide the game state.



```python
if not lost:
    move = ai.make_safe_move()
    if move is None:
        flags = ai.mines.copy()
```

Note that if there are no more safe moves, the situation itself has two possible conditions:

1. The remaining cells are all mines
2. The remaining cells may be safe or a mine, but AI can tell from the knowledge base.



```python
if lost:
    text = "Lost"
    l += 1
elif game.mines == flags:
    text = "Won"
    w += 1
elif flags != game.mines and len(flags) != 0:
    text = "Stuck"
    s += 1
```

As a result, if game.mines is the same as flags, which means that AI has mark all mines on the board, we can know that AI has won the game. On the other hand, if flags isn't the same as game.mines, which means that AI hasn't mark all mines on the board, and there is no more safe moves to make, the game becomes "stuck".

Result

When the number of initial safe cells are round(sqrt(#cells)):

Difficulty	easy	medium	hard
Win rate (%)	90	95	10

Here, the result of medium is higher than easy mode. I think that this is because the density of mines (#mines / #cells) are higher in easy mode, which means that mines are more likely to be arranged near each other, which makes AI harder to distinguish which cell is safe. In addition, the result of hard mode is very low, I would say that this is also because of the density of mines.

When the number of initial safe cells are (#cells)/10:

Difficulty	easy	medium	hard
Win rate (%)	95	95	25

Now, I raise the number of initial safe cells. This can make our AI get more clauses in the beginning. This time AI can get a 25% win rate in the hard mode.

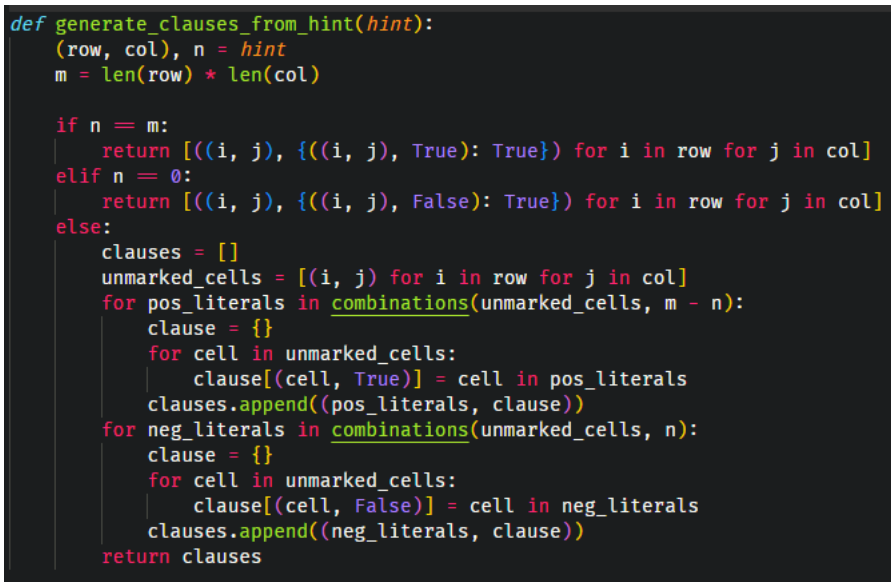
When the density of mines are (#cells)/10:

Difficulty	easy	medium	hard
Win rate (%)	91	94	93

This time I change the original density, making different difficulty has same mine density. As the chart show, win rates are very close between different difficulty. Therefore, I think instead of the size of the board, the density of mines are the main factor that affect AI's performance.

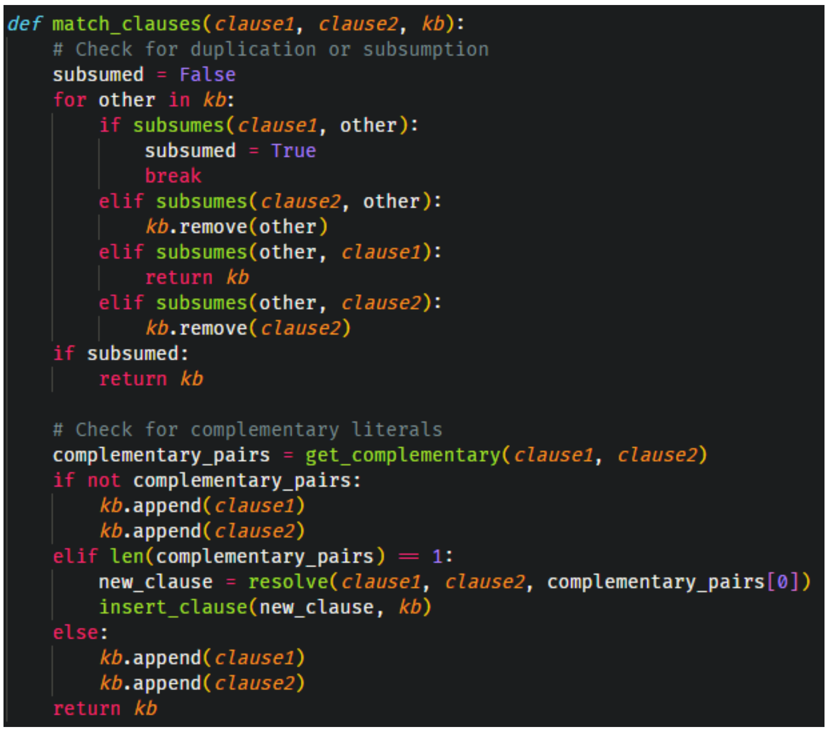
Other function

We can use hint to generate a new clause, which helps us to get new knowledge later.



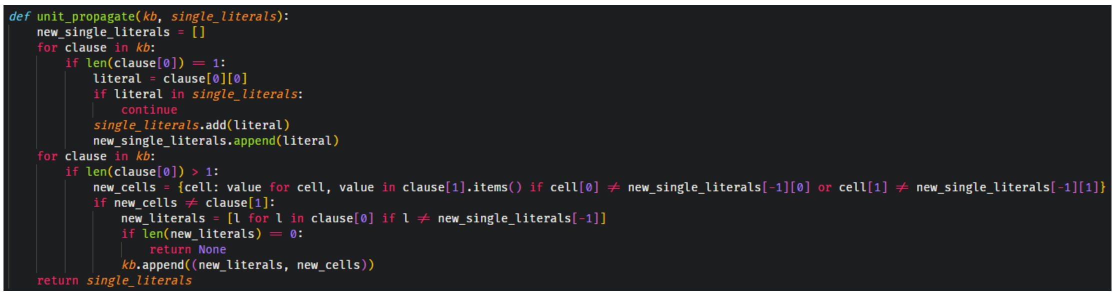
```python
def generate_clauses_from_hint(hint):
    (row, col), n = hint
    m = len(row) * len(col)

    if n == m:
        return [((i, j), {((i, j), True): True}) for i in row for j in col]
    elif n == 0:
        return [((i, j), {((i, j), False): True}) for i in row for j in col]
    else:
        clauses = []
        unmarked_cells = [(i, j) for i in row for j in col]
        for pos_literals in combinations(unmarked_cells, m - n):
            clause = {}
            for cell in unmarked_cells:
                clause[(cell, True)] = cell in pos_literals
            clauses.append((pos_literals, clause))
        for neg_literals in combinations(unmarked_cells, n):
            clause = {}
            for cell in unmarked_cells:
                clause[(cell, False)] = cell in neg_literals
            clauses.append((neg_literals, clause))
        return clauses
```

If the new clause we get is a single lateral clause, mark the cell (either safe or mine) and move that clause to KB. Next, no matter the incoming clause is single or not, we need to do matching, by which we can get some clearer clues in making safe steps. To begin with, check for duplication or subsumption. Next, check for complementary literals.



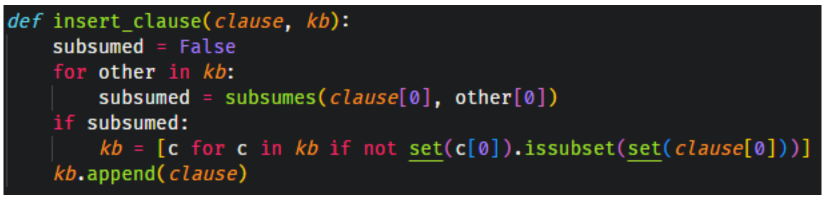
```python
def match_clauses(clause1, clause2, kb):
    # Check for duplication or subsumption
    subsumed = False
    for other in kb:
        if subsumes(clause1, other):
            subsumed = True
            break
        elif subsumes(clause2, other):
            kb.remove(other)
        elif subsumes(other, clause1):
            return kb
        elif subsumes(other, clause2):
            kb.remove(clause2)
    if subsumed:
        return kb

    # Check for complementary literals
    complementary_pairs = get_complementary(clause1, clause2)
    if not complementary_pairs:
        kb.append(clause1)
        kb.append(clause2)
    elif len(complementary_pairs) == 1:
        new_clause = resolve(clause1, clause2, complementary_pairs[0])
        insert_clause(new_clause, kb)
    else:
        kb.append(clause1)
        kb.append(clause2)
    return kb
```

Moreover, to avoid making our KB too large, we need to implement unit-propagation. By doing this, we can ensure that the clauses we maintained in KB are all clauses that cannot be resolve with other clauses. So, if a new single-literal clause appeared, discard those multi-literal clauses containing the incoming clause if both two clauses are positive or negative., and if not, we can pull out the incoming clause from that multi-literal clause.



```
def unit_propagate(kb, single_literals):
    new_single_literals = []
    for clause in kb:
        if len(clause[0]) == 1:
            literal = clause[0][0]
            if literal in single_literals:
                continue
            single_literals.add(literal)
            new_single_literals.append(literal)
    for clause in kb:
        if len(clause[0]) > 1:
            new_cells = {cell: value for cell, value in clause[1].items() if cell[0] ≠ new_single_literals[-1][0] or cell[1] ≠ new_single_literals[-1][1]}
            if new_cells ≠ clause[1]:
                new_literals = [l for l in clause[0] if l ≠ new_single_literals[-1]]
                if len(new_literals) == 0:
                    return None
                kb.append((new_literals, new_cells))
    return single_literals
```

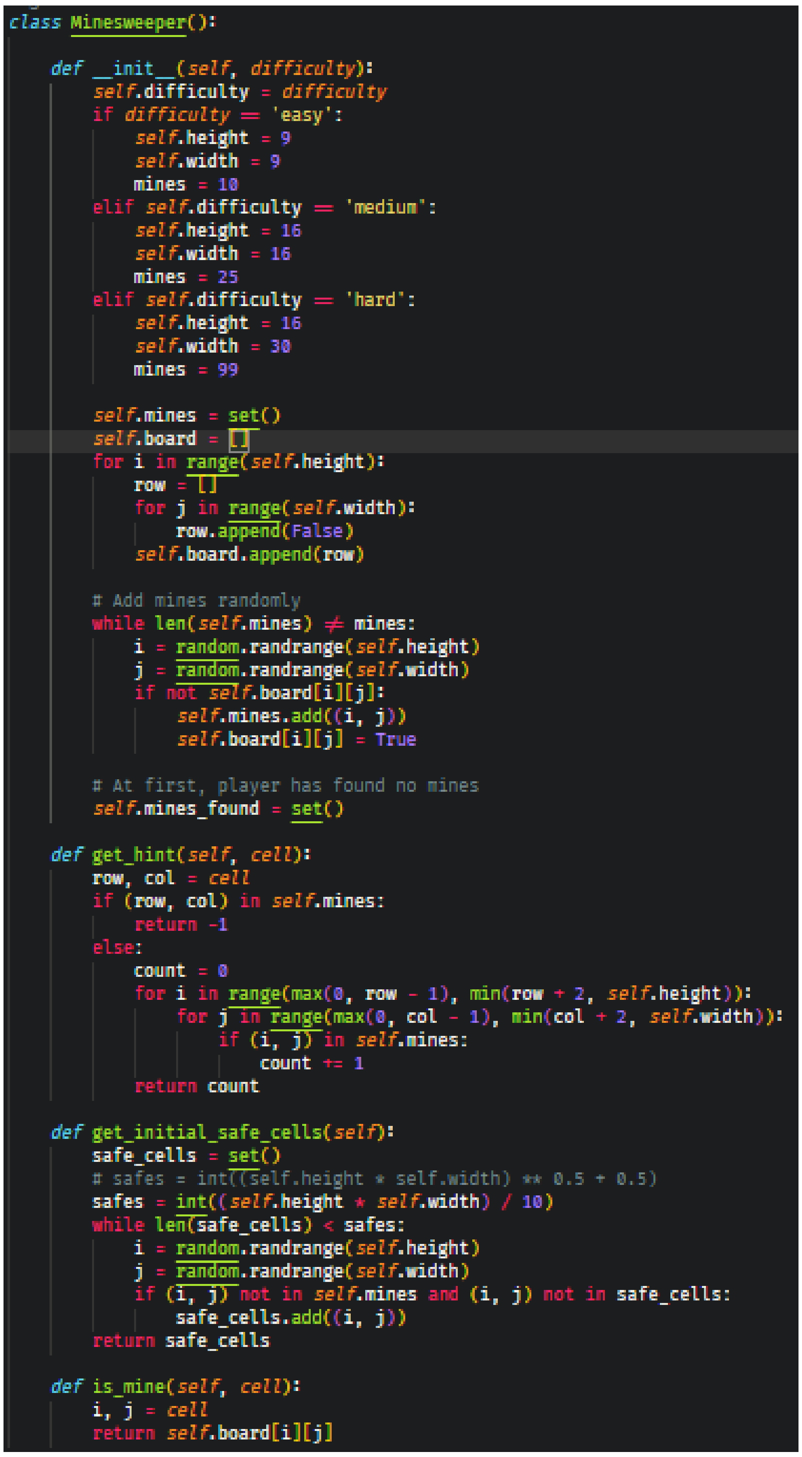
After we complete matching, under certain circumstances (mostly when new clause appears), we need to insert the clauses into KB. During the insertion, I first check if there the clause is identical. If not, I then start to check for subsumption.



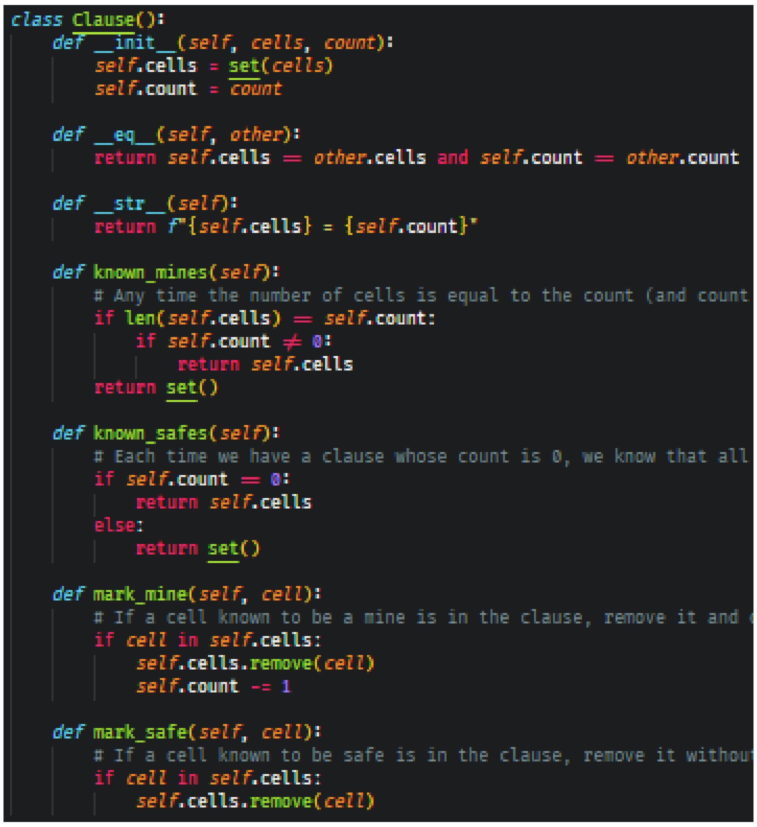
```
def insert_clause(clause, kb):
    subsumed = False
    for other in kb:
        subsumed = subsumes(clause[0], other[0])
    if subsumed:
        kb = [c for c in kb if not set(c[0]).issubset(set(clause[0]))]
    kb.append(clause)
```

Finally, we finish an iteration. And we can start over by making a new safe move, get new hint and clause, matching… etc.

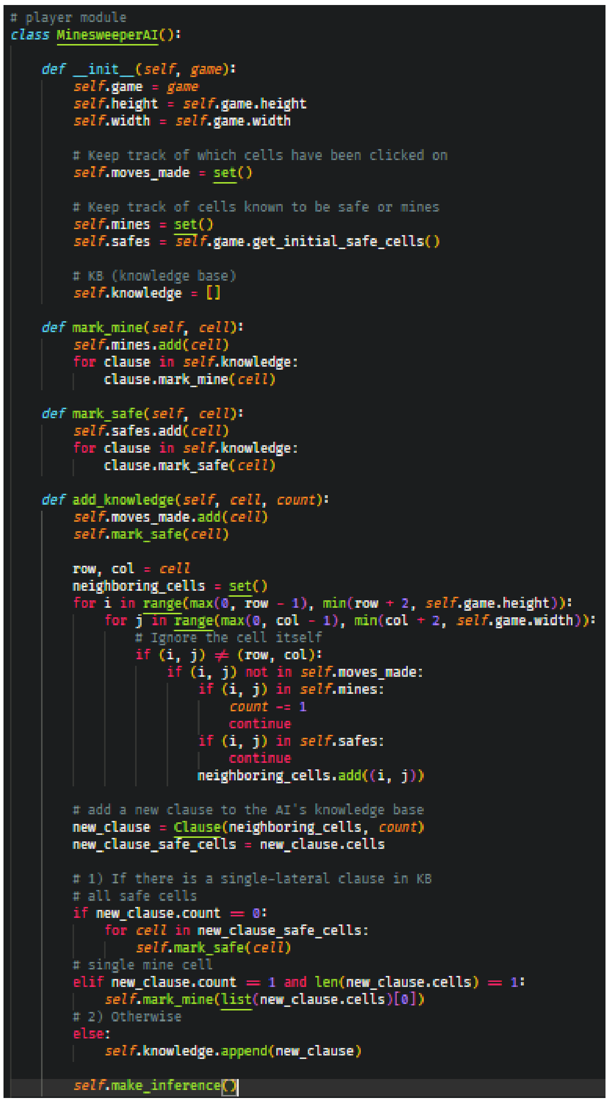
Appendix



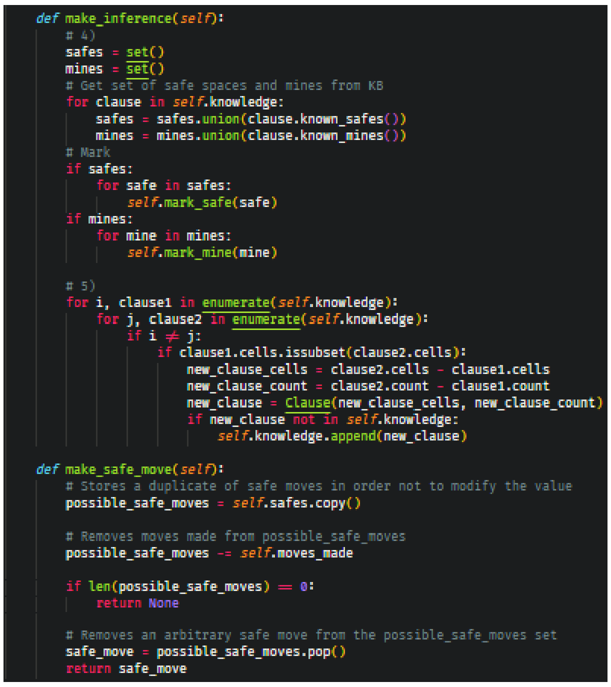
```python
class Minesweeper():

    def __init__(self, difficulty):
        self.difficulty = difficulty
        if difficulty == 'easy':
            self.height = 9
            self.width = 9
            mines = 10
        elif self.difficulty == 'medium':
            self.height = 16
            self.width = 16
            mines = 25
        elif self.difficulty == 'hard':
            self.height = 16
            self.width = 30
            mines = 99

        self.mines = set()
        self.board = []
        for i in range(self.height):
            row = []
            for j in range(self.width):
                row.append(False)
            self.board.append(row)

        # Add mines randomly
        while len(self.mines) != mines:
            i = random.randrange(self.height)
            j = random.randrange(self.width)
            if not self.board[i][j]:
                self.mines.add((i, j))
                self.board[i][j] = True

        # At first, player has found no mines
        self.mines_found = set()

    def get_hint(self, cell):
        row, col = cell
        if (row, col) in self.mines:
            return -1
        else:
            count = 0
            for i in range(max(0, row - 1), min(row + 2, self.height)):
                for j in range(max(0, col - 1), min(col + 2, self.width)):
                    if (i, j) in self.mines:
                        count += 1
            return count

    def get_initial_safe_cells(self):
        safe_cells = set()
        # safes = int((self.height * self.width) ** 0.5 + 0.5)
        safes = int((self.height * self.width) / 10)
        while len(safe_cells) < safes:
            i = random.randrange(self.height)
            j = random.randrange(self.width)
            if (i, j) not in self.mines and (i, j) not in safe_cells:
                safe_cells.add((i, j))
        return safe_cells

    def is_mine(self, cell):
        i, j = cell
        return self.board[i][j]
```



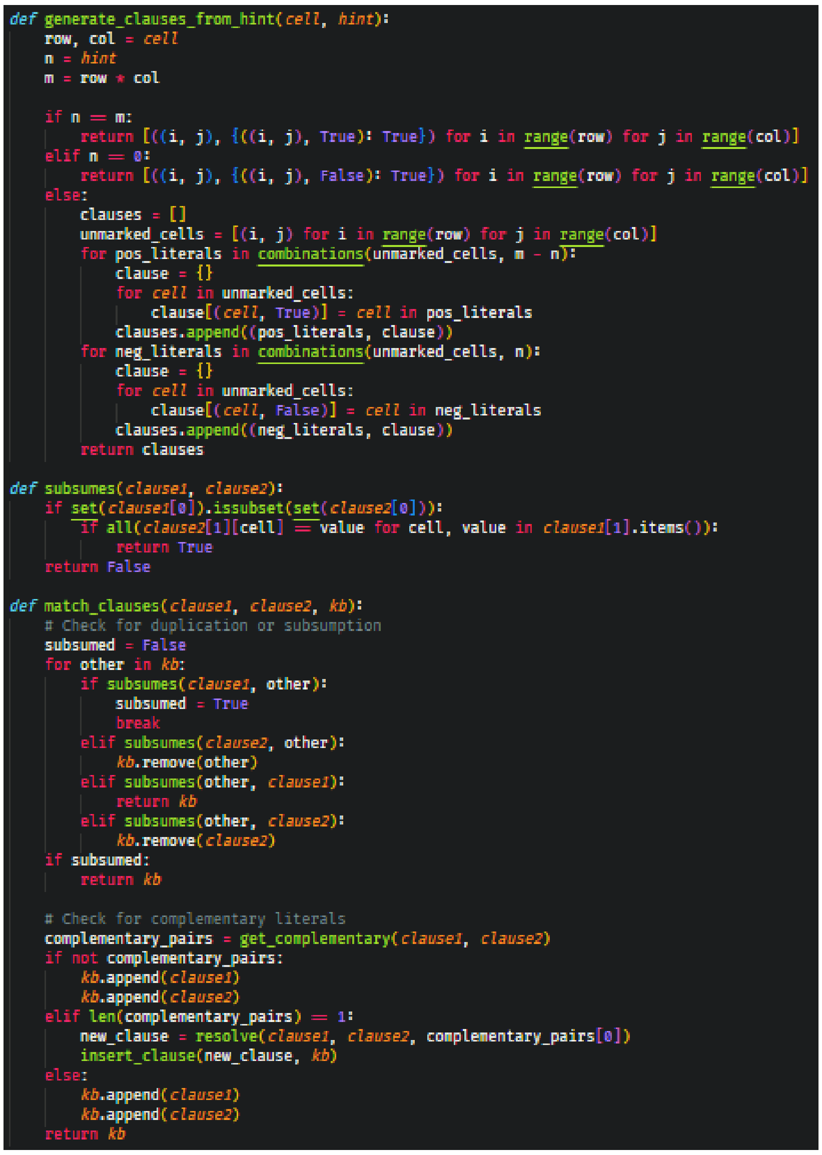
```python
class Clause():
    def __init__(self, cells, count):
        self.cells = set(cells)
        self.count = count

    def __eq__(self, other):
        return self.cells == other.cells and self.count == other.count

    def __str__(self):
        return f"{self.cells} = {self.count}"

    def known_mines(self):
        # Any time the number of cells is equal to the count (and count
        if len(self.cells) == self.count:
            if self.count != 0:
                return self.cells
        return set()

    def known_safes(self):
        # Each time we have a clause whose count is 0, we know that all
        if self.count == 0:
            return self.cells
        else:
            return set()

    def mark_mine(self, cell):
        # If a cell known to be a mine is in the clause, remove it and
        if cell in self.cells:
            self.cells.remove(cell)
            self.count -= 1

    def mark_safe(self, cell):
        # If a cell known to be safe is in the clause, remove it withou
        if cell in self.cells:
            self.cells.remove(cell)
```



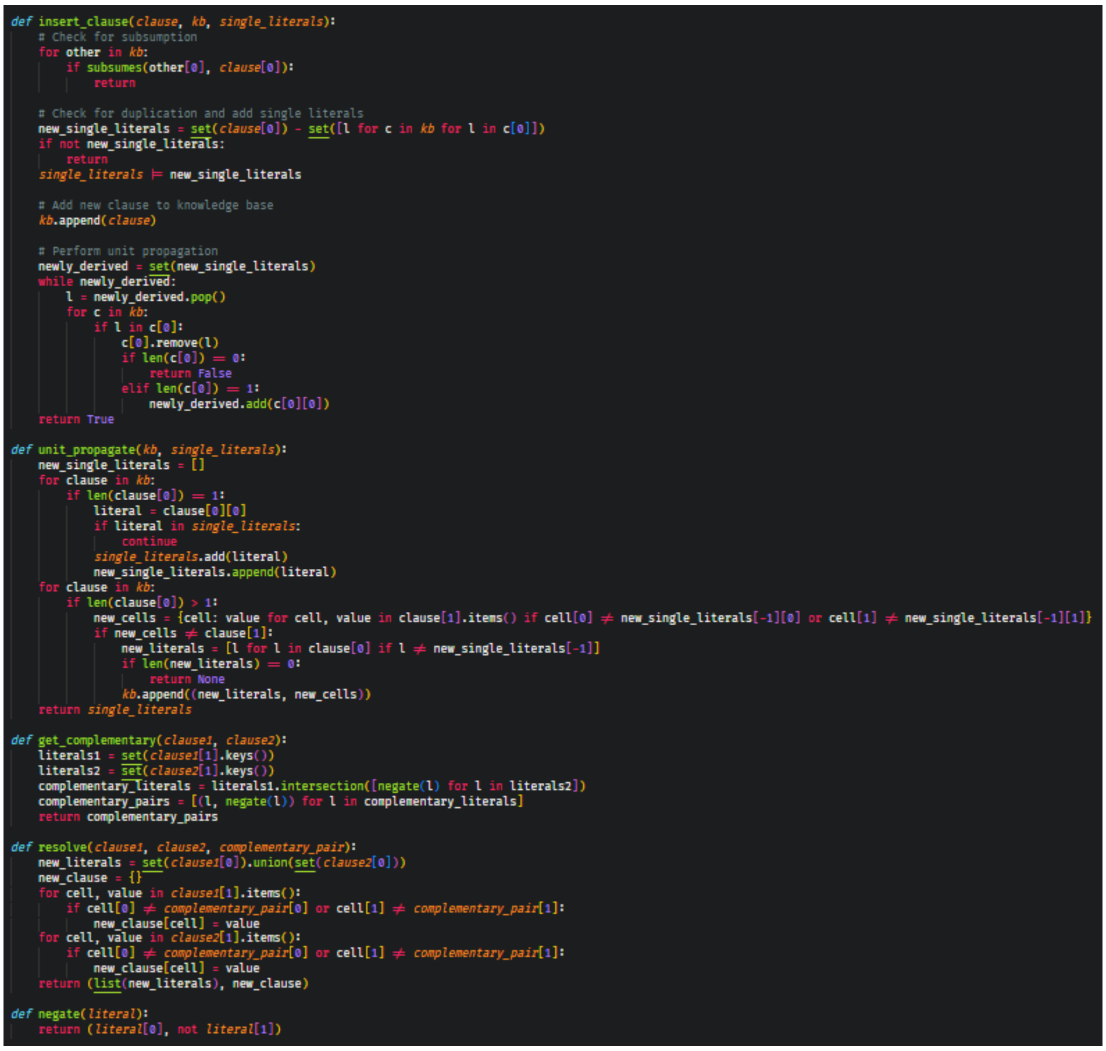
```python
# player module
class MinesweeperAI():

    def __init__(self, game):
        self.game = game
        self.height = self.game.height
        self.width = self.game.width

        # Keep track of which cells have been clicked on
        self.moves_made = set()

        # Keep track of cells known to be safe or mines
        self.mines = set()
        self.safes = self.game.get_initial_safe_cells()

        # KB (knowledge base)
        self.knowledge = []

    def mark_mine(self, cell):
        self.mines.add(cell)
        for clause in self.knowledge:
            clause.mark_mine(cell)

    def mark_safe(self, cell):
        self.safes.add(cell)
        for clause in self.knowledge:
            clause.mark_safe(cell)

    def add_knowledge(self, cell, count):
        self.moves_made.add(cell)
        self.mark_safe(cell)

        row, col = cell
        neighboring_cells = set()
        for i in range(max(0, row - 1), min(row + 2, self.game.height)):
            for j in range(max(0, col - 1), min(col + 2, self.game.width)):
                # Ignore the cell itself
                if (i, j) ≠ (row, col):
                    if (i, j) not in self.moves_made:
                        if (i, j) in self.mines:
                            count -= 1
                            continue
                        if (i, j) in self.safes:
                            continue
                        neighboring_cells.add((i, j))

        # add a new clause to the AI's knowledge base
        new_clause = Clause(neighboring_cells, count)
        new_clause_safe_cells = new_clause.cells

        # 1) If there is a single-lateral clause in KB
        # all safe cells
        if new_clause.count == 0:
            for cell in new_clause_safe_cells:
                self.mark_safe(cell)
        # single mine cell
        elif new_clause.count == 1 and len(new_clause.cells) == 1:
            self.mark_mine(list(new_clause.cells)[0])
        # 2) Otherwise
        else:
            self.knowledge.append(new_clause)

        self.make_inference()
```



```python
    def make_inference(self):
        # 4)
        safes = set()
        mines = set()
        # Get set of safe spaces and mines from KB
        for clause in self.knowledge:
            safes = safes.union(clause.known_safes())
            mines = mines.union(clause.known_mines())
        # Mark
        if safes:
            for safe in safes:
                self.mark_safe(safe)
        if mines:
            for mine in mines:
                self.mark_mine(mine)

        # 5)
        for i, clause1 in enumerate(self.knowledge):
            for j, clause2 in enumerate(self.knowledge):
                if i ≠ j:
                    if clause1.cells.issubset(clause2.cells):
                        new_clause_cells = clause2.cells - clause1.cells
                        new_clause_count = clause2.count - clause1.count
                        new_clause = Clause(new_clause_cells, new_clause_count)
                        if new_clause not in self.knowledge:
                            self.knowledge.append(new_clause)

    def make_safe_move(self):
        # Stores a duplicate of safe moves in order not to modify the value
        possible_safe_moves = self.safes.copy()

        # Removes moves made from possible_safe_moves
        possible_safe_moves -= self.moves_made

        if len(possible_safe_moves) == 0:
            return None

        # Removes an arbitrary safe move from the possible_safe_moves set
        safe_move = possible_safe_moves.pop()
        return safe_move
```



```python
def generate_clauses_from_hint(cell, hint):
    row, col = cell
    n = hint
    m = row * col

    if n == m:
        return [((i, j), {((i, j), True): True}) for i in range(row) for j in range(col)]
    elif n == 0:
        return [((i, j), {((i, j), False): True}) for i in range(row) for j in range(col)]
    else:
        clauses = []
        unmarked_cells = [(i, j) for i in range(row) for j in range(col)]
        for pos_literals in combinations(unmarked_cells, m - n):
            clause = {}
            for cell in unmarked_cells:
                clause[(cell, True)] = cell in pos_literals
            clauses.append((pos_literals, clause))
        for neg_literals in combinations(unmarked_cells, n):
            clause = {}
            for cell in unmarked_cells:
                clause[(cell, False)] = cell in neg_literals
            clauses.append((neg_literals, clause))
        return clauses

def subsumes(clause1, clause2):
    if set(clause1[0]).issubset(set(clause2[0])):
        if all(clause2[1][cell] == value for cell, value in clause1[1].items()):
            return True
    return False

def match_clauses(clause1, clause2, kb):
    # Check for duplication or subsumption
    subsumed = False
    for other in kb:
        if subsumes(clause1, other):
            subsumed = True
            break
        elif subsumes(clause2, other):
            kb.remove(other)
        elif subsumes(other, clause1):
            return kb
        elif subsumes(other, clause2):
            kb.remove(clause2)
    if subsumed:
        return kb

    # Check for complementary literals
    complementary_pairs = get_complementary(clause1, clause2)
    if not complementary_pairs:
        kb.append(clause1)
        kb.append(clause2)
    elif len(complementary_pairs) == 1:
        new_clause = resolve(clause1, clause2, complementary_pairs[0])
        insert_clause(new_clause, kb)
    else:
        kb.append(clause1)
        kb.append(clause2)
    return kb
```



```python
def insert_clause(clause, kb, single_literals):
    # Check for subsumption
    for other in kb:
        if subsumes(other[0], clause[0]):
            return

    # Check for duplication and add single literals
    new_single_literals = set(clause[0]) - set([l for c in kb for l in c[0]])
    if not new_single_literals:
        return
    single_literals |= new_single_literals

    # Add new clause to knowledge base
    kb.append(clause)

    # Perform unit propagation
    newly_derived = set(new_single_literals)
    while newly_derived:
        l = newly_derived.pop()
        for c in kb:
            if l in c[0]:
                c[0].remove(l)
                if len(c[0]) == 0:
                    return False
                elif len(c[0]) == 1:
                    newly_derived.add(c[0][0])
    return True

def unit_propagate(kb, single_literals):
    new_single_literals = []
    for clause in kb:
        if len(clause[0]) == 1:
            literal = clause[0][0]
            if literal in single_literals:
                continue
            single_literals.add(literal)
            new_single_literals.append(literal)
    for clause in kb:
        if len(clause[0]) > 1:
            new_cells = {cell: value for cell, value in clause[1].items() if cell[0] != new_single_literals[-1][0] or cell[1] != new_single_literals[-1][1]}
            if new_cells != clause[1]:
                new_literals = [l for l in clause[0] if l != new_single_literals[-1]]
                if len(new_literals) == 0:
                    return None
                kb.append((new_literals, new_cells))
    return single_literals

def get_complementary(clause1, clause2):
    literals1 = set(clause1[1].keys())
    literals2 = set(clause2[1].keys())
    complementary_literals = literals1.intersection([negate(l) for l in literals2])
    complementary_pairs = [(l, negate(l)) for l in complementary_literals]
    return complementary_pairs

def resolve(clause1, clause2, complementary_pair):
    new_literals = set(clause1[0]).union(set(clause2[0]))
    new_clause = {}
    for cell, value in clause1[1].items():
        if cell[0] != complementary_pair[0] or cell[1] != complementary_pair[1]:
            new_clause[cell] = value
    for cell, value in clause2[1].items():
        if cell[0] != complementary_pair[0] or cell[1] != complementary_pair[1]:
            new_clause[cell] = value
    return (list(new_literals), new_clause)

def negate(literal):
    return (literal[0], not literal[1])
```