

# AI Capstone HW3 - Minesweeper

---

## Algorithm

### Propositional Logic

Propositional logic is a branch of mathematical logic that deals with the study of propositions, or statements, that are either true or false. It is a formal language that allows us to reason about the truth or falsity of such statements using logical operators, such as "and", "or", "not", and "implies".

### The Rule of Resolution

The rule of resolution is a fundamental inference rule in propositional logic used to derive new propositions from existing ones. It involves finding two clauses, each containing complementary literals, and using them to generate a new clause that is the union of the non-complementary literals.

### Conjunctive Normal Form (CNF)

The application of the resolution rule requires the KB to be in conjunctive normal form (CNF):

- CNF: The whole KB becomes the conjunction (AND) of a set of clauses.
- clauses: disjunctions (OR) of literals, or individual literals
- literals: atomic sentences or their negations
- CNF is combined by clauses, while a clause is combined by literals

## Implementation

### MineSweeperPlayer class

The MineSweeperPlayer class is the main class that implements the AI agent. It has the following attributes:

- KB: A set representing the knowledge base of the AI agent. The knowledge base contains Conjunctive Normal Form (CNF) clauses.
- KB0: A set representing the list of single-literal clauses representing marked cells.
- game: An instance of the MinesweeperGame class representing the Minesweeper game.
- initial\_safe\_cells: A set containing the initial safe cells in the game.
- KB\_length\_history: A list containing the length of the knowledge base at each move.

### subsumption\_checking

- Check for subsumption between two clauses:
  - An example of subsumption: (x2 or x3) is stricter than (x1 or x2 or x3): The former entails the latter. As a result, we do not need the less strict clause anymore.
    - clause1 is stricter than an clause2: return clause1
    - clause2 is stricter than an clause1: return clause2
    - else if there is no subsumption: return None

### resolve

Do resolution between two clauses using the resolution rule Resolution rule:

- If two clauses have complementary literals:
  - If there is only one pair of complementary literals:
    - Apply resolution to generate a new clause.
    - Resolution applying: Remove the pair of complementary literals from the two clauses, and union the two clauses.
    - Example:  $(x_1 \text{ or } x_2) \text{ and } (\text{not } x_1 \text{ or } x_3) \rightarrow (x_2 \text{ or } x_3)$
  - If there are more than one pairs of complementary literals:
    - Do nothing here. (Resolution will results in tautology (always true).)
    - Example:  $(\text{not } x_2 \text{ or } x_3) \text{ and } (x_1 \text{ or } x_2 \text{ or } \text{not } x_3)$

## matching\_clauses

About "matching" two clauses:

- Check for duplication or subsumption first. Keep only the more strict clause.
- If no duplication or subsumption, do resolution.

## unit\_propagation

Unit-propagation heuristic:

- If a clause has only one literal A:
  - For each multi-literal clause in KB containing A:
    - If the two occurrences of A are both positive or both negative:
      - Discard the multi-literal clause. It is always true. (This is a case of subsumption.)
    - Else
      - Remove A from the multi-literal clause. This is the result of resolution.
- Else if a clause has more than one literal:
  - Check against the single-literal clauses in KB0.

## insert\_clause\_to\_KB

About inserting a new clause to the KB:

- Skip the insertion if there is an identical clause in KB or KB0.
- Do unit-propagation with the new clause.
- Do resolution of the new clause with all the clauses in KB0, if applicable. Keep only the resulting clause.
- Check for subsumption with all the clauses in KB:
  - New clause is stricter than an existing clause: Delete the existing clause.
  - An existing clause is stricter than the new clause): Skip (no insertion).

## generate\_clauses

About generating clauses from the hints:

- Each hint provides the following information: There are n mines in a list of m unmarked cells.
- $(n == m)$ : Insert the m single-literal positive clauses to the KB, one for each unmarked cell.

- ( $n == 0$ ): Insert the  $m$  single-literal negative clauses to the KB, one for each unmarked cell.
- ( $m > n > 0$ ): General cases (need to generate CNF clauses and add them to the KB):
  - $C(m, m-n+1)$  clauses, each having  $m-n+1$  positive literals
  - $C(m, n+1)$  clauses, each having  $n+1$  negative literals.
  - For example, for  $m=5$  and  $n=2$ , let the cells be  $x_1, x_2, \dots, x_5$ : There are  $C(5,4)$  all-positive-literal clauses:  $(x_1 \text{ or } x_2 \text{ or } x_3 \text{ or } x_4)$ ,  $(x_1 \text{ or } x_2 \text{ or } x_3 \text{ or } x_5)$ , ...,  $(x_2 \text{ or } x_3 \text{ or } x_4 \text{ or } x_5)$  There are  $C(5,3)$  all-negative-literal clauses:  $(\text{not } x_1 \text{ or not } x_2 \text{ or not } x_3)$ ,  $(\text{not } x_1 \text{ or not } x_2 \text{ or not } x_4)$ ,  $(\text{not } x_1 \text{ or not } x_2 \text{ or not } x_5)$ , ...,  $(\text{not } x_3 \text{ or not } x_4 \text{ or not } x_5)$

### is\_stuck

Determine if the game is stucked by KB length history. If the KB length is the same for 5 consecutive turns, then the game is stucked.

### make\_safe\_guess

Make a guess if the game is stucked. Choose a guess by one of the literal in the clause with the length of 2 in KB. If there's no clause with the length of 2, then change that to 3, 4, 5, ...

### game\_move

Perform one iteration of the game.

- Check if the game is stucked, if so, make a safe guess
  - If the guess isn't safe, then the game is over
  - Else, mark the guess as safe and continue the game
- If there is a single-literal clause in the KB:
  - If this cell is safe:
    - Mark that cell as safe or mined on the game board.
    - Query the game control module for the hint at that cell.
    - Generate new clauses from the hint.
- Else:
  - Apply pairwise "matching" of the clauses in the KB or apply pairwise "matching" of the clauses between KB and KB0.

### pairwise\_matching

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.

### pairwise\_matching\_KB0

Apply pairwise "matching" of the clauses between KB and KB0.

- If new clauses are generated due to resolution, insert them into the KB.

## Experiments

MineSweeperGame class Explanation:

MineSweeperGame class can create a Minesweeper game, randomly generate mines on the game board, and create a graphical user interface using `pygame`. It contains various functions for marking and revealing cells on the board, and for counting adjacent mines to a given cell. Additionally, there is a function for getting the initial safe cells on the game board, as well as a function for drawing the game board. After every step, it will determine if the AI agent is win or lose.

For this game, there are three three difficulty levels:

- Easy (9x9 board with 10 mines)
- Medium (16x16 board with 25 mines)
- Hard (30x16 board with 99 mines.)

Moreover, it will provide an initial list of "safe" cells.

- In this assignment, I use `round(sqrt(#cells)) * initial_safe_cells_times` as the number of initial safe cells.

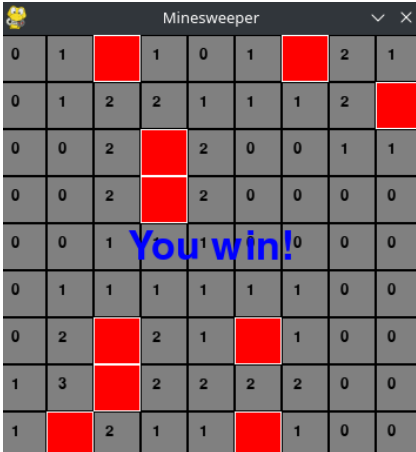
Testing (initial\_safe\_cells\_times = 1)

Level Easy

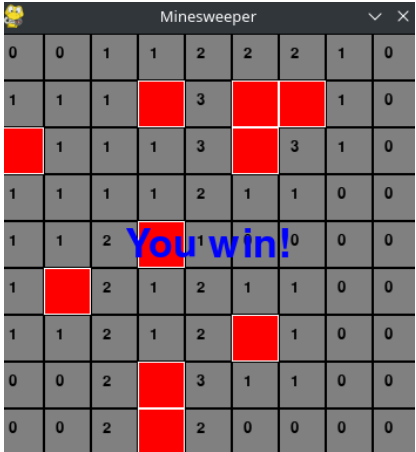
For level `easy`, AI agent can almost win all the game in one second. For playing 10 games, it has won 9 games.

- Win

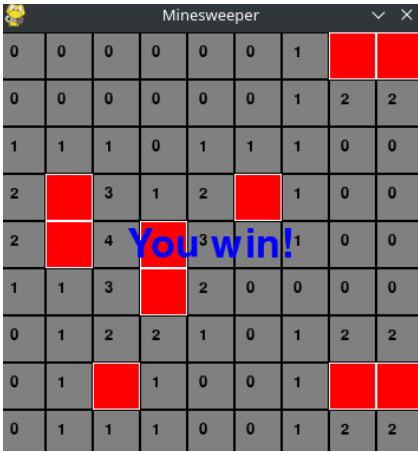
Experiment 1



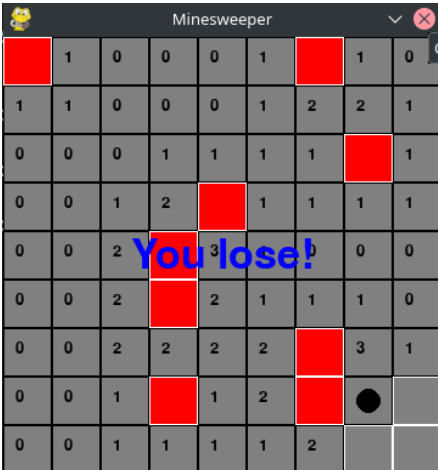
Experiment 2



Experiment 3



- Lose

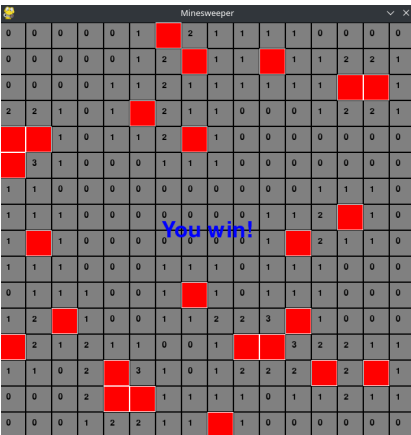


Level Medium

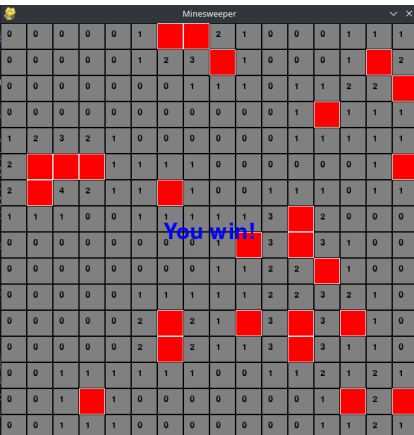
For level **medium**, AI agent can almost win all the game in five second. For playing 10 games, it has won 9 games.

- Win

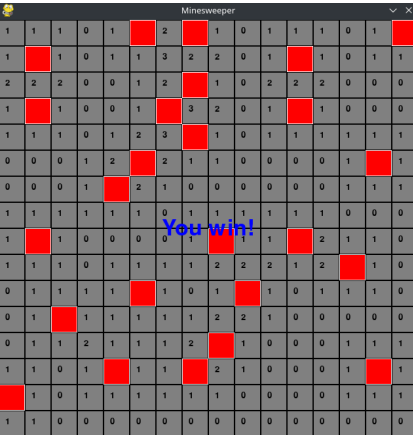
Experiment 1



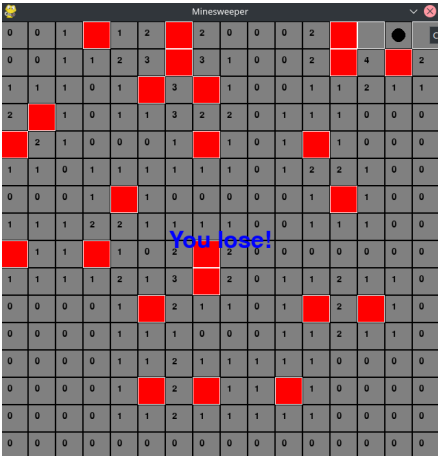
Experiment 2



Experiment 3



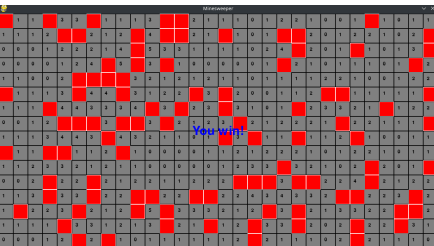
- Lose



Level Hard

For level **hard**, AI agent will finish the game at most 30 second. For playing 10 games, it has only won 1 games.

- Win

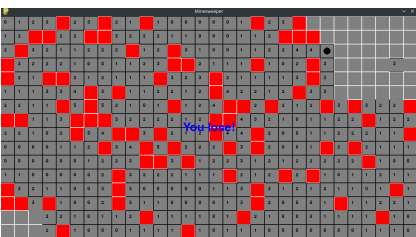
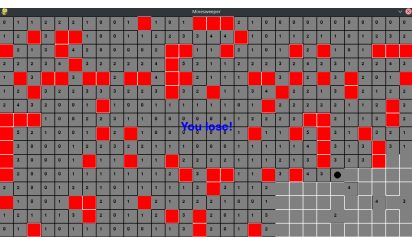
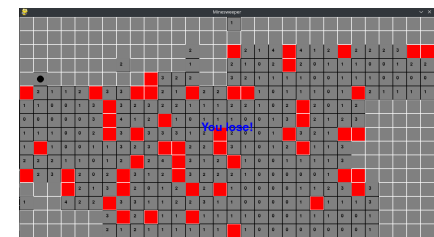


- Lose

Experiment 1

Experiment 2

Experiment 3



Testing (initial\_safe\_cells\_times = 5)

Level Easy

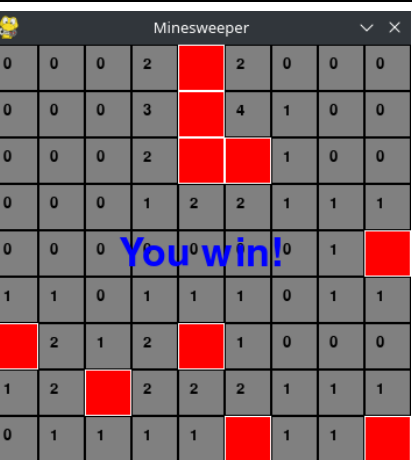
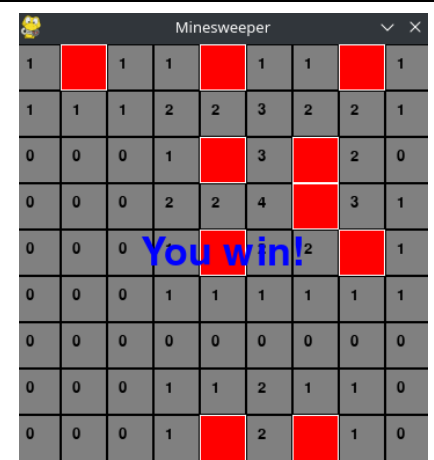
For level **easy**, AI agent can always win the game in one second. For playing 10 games, it has won 10 games.

- Win

Experiment 1

Experiment 2

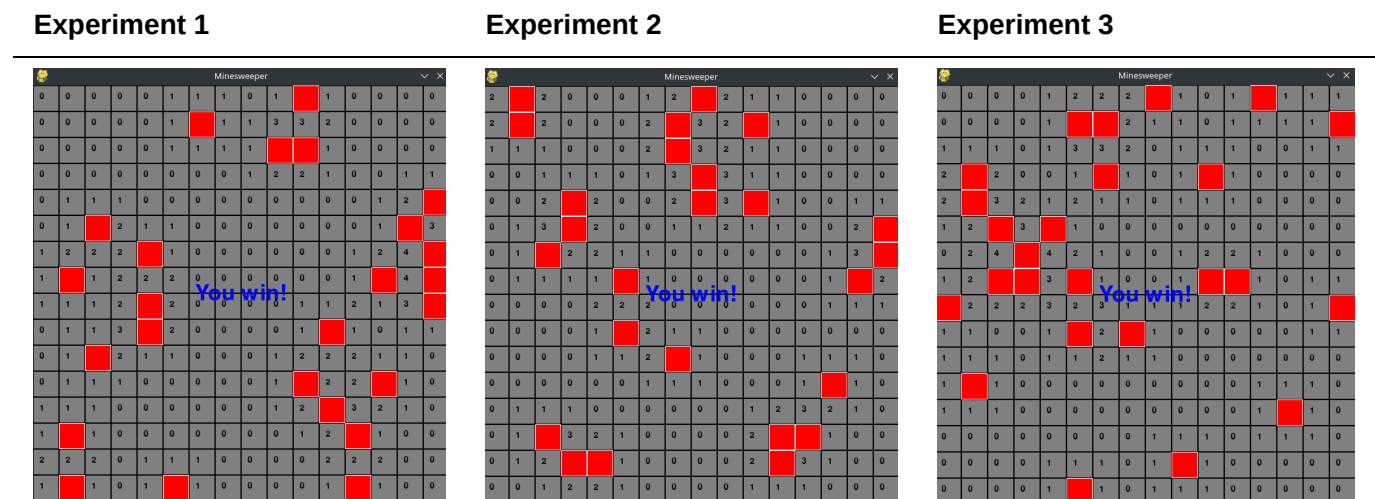
Experiment 3



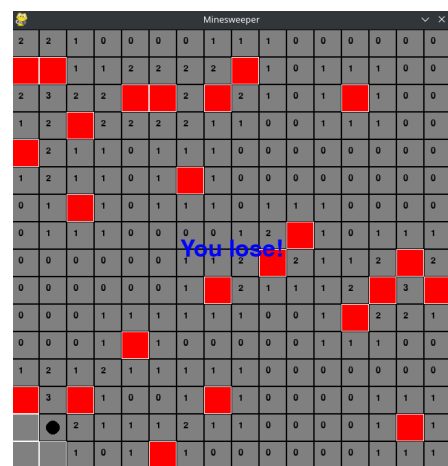
Level Medium

For level **medium**, AI agent can almost win all the game in five second. For playing 10 games, it has won 9 games.

- Win



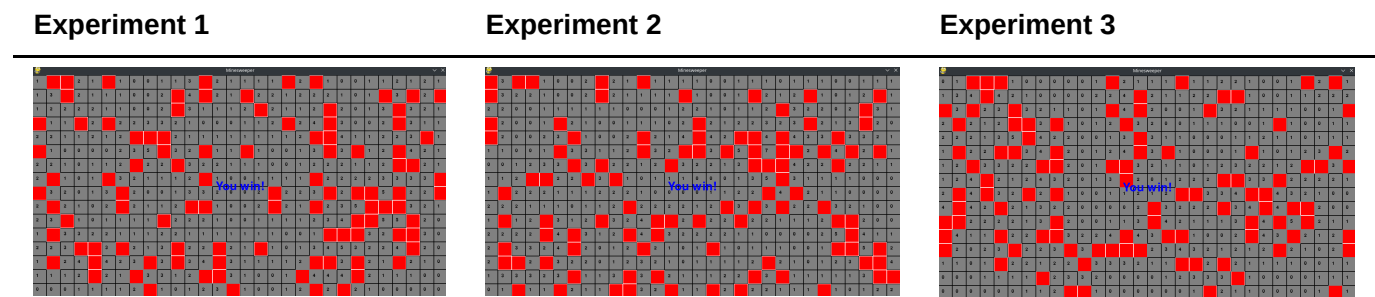
- Lose



Level Hard

For level **hard**, AI agent will finish the game at most 25 second. For playing 10 games, it has won 4 games.

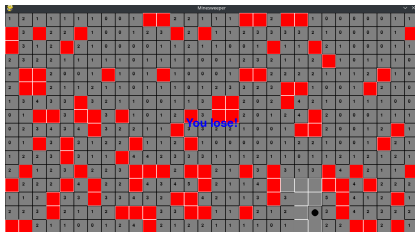
- Win



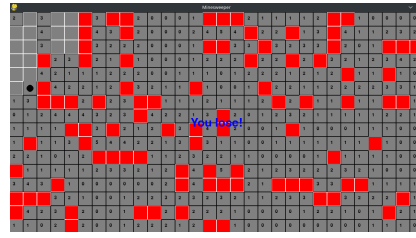
- Lose



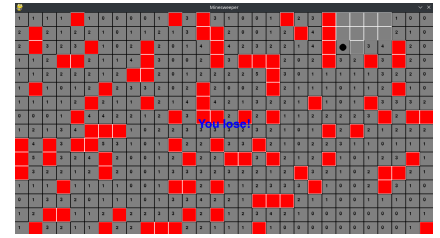
## Experiment 1



## Experiment 2



## Experiment 3



Testing (initial\_safe\_cells\_times = 10)

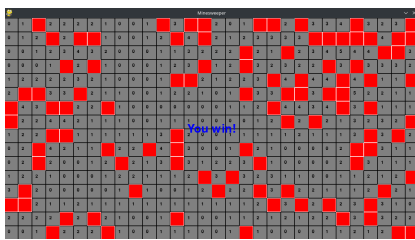
I only tested level hard because the win rate of other two levels are already high.

## Level Hard

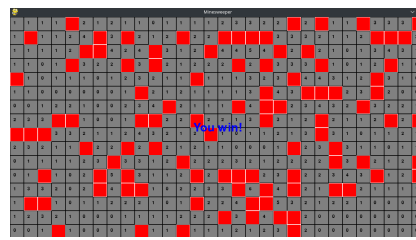
For level **hard**, AI agent will finish the game at most 20 second. For playing 10 games, it has won 7 games.

- Win

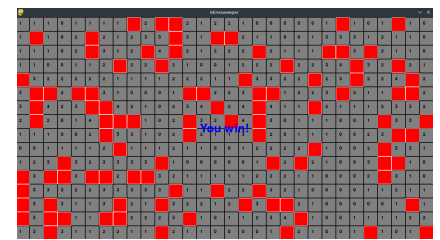
## Experiment 1



## Experiment 2

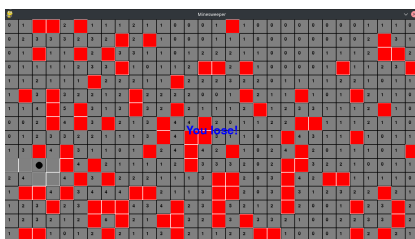


## Experiment 3

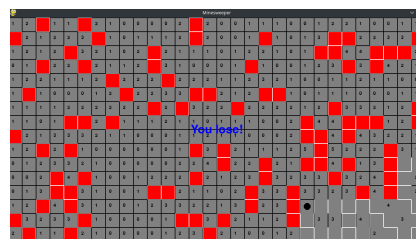


- Lose

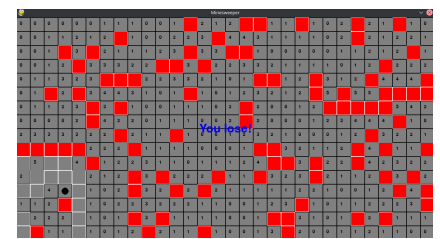
## Experiment 1



## Experiment 2



## Experiment 3



## Discussion

- From all the experiment and results above, I discovered something interesting:
  - For the difficulty level, hard is much harder than medium and easy. In experiments, the win rate of level hard is a lot lower than the other two levels. Additionally, it takes more time for level hard to run than the other two levels.
  - For **initial\_safe\_cells\_times**, it affects the win rate of hard level a lot. During the change of **initial\_safe\_cells\_times** = 1 -> 5 -> 10, the win rate of hard level increases from 1/10 -> 4/10 -> 7/10, and the time to run has decreased, too.
  - For pairwise matching method, I choose **pairwise\_matching\_KB0** function in default, which apply pairwise "matching" of the clauses between KB and KB0, and it performs well. It only cost few time for every step to move. However, if I choose only **pairwise\_matching**, which apply



pairwise "matching" of the clauses in the KB, or choose both `pairwise_matching` and `pairwise_matching_KB0`, it will cost a lot of time for every step to move. So I choose `pairwise_matching_KB0` at last.

## Appendix

`MineSweeperAI.py`:

```
import itertools

class MineSweeperPlayer:
    def __init__(self, game):
        # A set representing the knowledge base of the AI agent.
        # The knowledge base contains Conjunctive Normal Form (CNF)
        clauses.
        self.KB = set()
        # A set representing the list of single-literal clauses
        representing marked cells.
        self.KB0 = set()
        # An instance of the MinesweeperGame class representing the
        Minesweeper game.
        self.game = game
        # A set containing the initial safe cells in the game.
        self.initial_safe_cells = self.game.get_initial_safe_cells()
        # A list containing the length of the knowledge base at each step.
        self.KB_length_history = []

        # Add negative single-literal clauses for initial safe cells to the
        KB
        for cell in self.initial_safe_cells:
            # print(f"Initial safe cell: {cell}")
            self.KB.add(((cell[0], cell[1], False),))

    def mark_safe(self, literal):
        """
        Mark a cell as safe
        """
        self.game.mark_safe(literal[0], literal[1])

    def mark_mine(self, literal):
        """
        Mark a cell as mine
        """
        self.game.mark_mine(literal[0], literal[1])

    def get_unmarked_neighbors_and_mine_count(self, literal):
        """
        Get unmarked neighbors and mine count of a cell
        """
        neighbors, mine_count =
self.game.get_unmarked_neighbors_and_mine_count(literal[0], literal[1])
        return neighbors, mine_count
```

```

def subsumption_checking(self, clause1, clause2):
    '''
    - Check for subsumption between two clauses:
        - An example of subsumption:
            (x2 or x3) is stricter than (x1 or x2 or x3): The former
entails the latter.
            As a result, we do not need the less strict clause anymore.
        - clause1 is stricter than an clause2: return clause1
        - clause2 is stricter than an clause1: return clause2
        - else if there is no subsumption: return None
    '''
    clause1 = set(clause1)
    clause2 = set(clause2)
    if clause1.issubset(clause2):
        return tuple(clause1)
    elif clause2.issubset(clause1):
        return tuple(clause2)
    else:
        return None

def resolve(self, clause1, clause2):
    '''
    Do resolution between two clauses using the resolution rule
    Resolution rule:
    - If two clauses have complementary literals:
        - If there is only one pair of complementary literals:
            - Apply resolution to generate a new clause.
            - Resolution applying: Remove the pair of complementary
literals from the two clauses, and union the two clauses.
            - Example: (x1 or x2) and (not x1 or x3) -> (x2 or x3)
        - If there are more than one pairs of complementary literals:
            - Do nothing here. (Resolution will results in tautology
(always true).)
            - Example: (not x2 or x3) and (x1 or x2 or not x3)
    '''
    clause1 = set(clause1)
    clause2 = set(clause2)

    literal_to_remove = None
    complementary_literals_cnt = 0
    for literal1 in clause1:
        if (literal1[0], literal1[1], not literal1[2]) in clause2:
            literal_to_remove = literal1
            complementary_literals_cnt += 1

    if literal_to_remove and complementary_literals_cnt == 1:
        clause1.remove(literal_to_remove)
        clause2.remove((literal_to_remove[0], literal_to_remove[1], not
literal_to_remove[2]))
        return tuple(clause1.union(clause2))
    return None

def matching_clauses(self, clause1, clause2):

```

```

'''
    About "matching" two clauses:
    - Check for duplication or subsumption first. Keep only the more
strict clause.
    - If no duplication or subsumption, do resolution.
'''
    clause = self.subsumption_checking(clause1, clause2)
    is_strict = False
    if clause:
        is_strict = True
    else:
        clause = self.resolve(clause1, clause2)
    return clause, is_strict

def unit_propagation(self, clause):
    '''
    Unit-propagation heuristic:
    - If a clause has only one literal A:
        - For each multi-literal clause in KB containing A:
            - If the two occurrences of A are both positive or both
negative:
                - Discard the multi-literal clause. It is always true.
(This is a case of subsumption.)
            - Else
                - Remove A from the multi-literal clause. This is the
result of resolution.
        - Else if a clause has more than one literal:
            - Check against the single-literal clauses in KB0.
    '''
    clauses_to_add = set()
    clauses_to_remove = set()
    if len(clause) == 1:
        for clause2 in self.KB:
            if clause[0] in clause2:
                clauses_to_remove.add(clause2)
            elif (clause[0][0], clause[0][1], not clause[0][2]) in
clause2:
                new_clause = tuple(literal for literal in clause2 if
literal != (clause[0][0], clause[0][1], not clause[0][2]))
                clauses_to_add.add(new_clause)
                clauses_to_remove.add(clause2)
    else:
        for clause0 in self.KB0:
            if clause0[0] in clause:
                break
            elif (clause0[0][0], clause0[0][1], not clause0[0][2]) in
clause:
                new_clause = tuple(literal for literal in clause if
literal != (clause0[0][0], clause0[0][1], not clause0[0][2]))
                clauses_to_add.add(new_clause)
        for clause_to_add in clauses_to_add:
            self.KB.add(clause_to_add)
        for clause_to_remove in clauses_to_remove:
            self.KB.remove(clause_to_remove)

```

```

def insert_clause_to_KB(self, clause):
    """
    About inserting a new clause to the KB:
    - Skip the insertion if there is an identical clause in KB or KB0.
    - Do unit-propagation with the new clause.
    - Do resolution of the new clause with all the clauses in KB0, if
    applicable. Keep only the resulting clause.
    - Check for subsumption with all the clauses in KB:
        - New clause is stricter than an existing clause: Delete the
    existing clause.
        - An existing clause is stricter than the new clause): Skip (no
    insertion).
    """
    if clause in self.KB or clause in self.KB0:
        return None

    self.unit_propagation(clause)
    # Do resolution with all the clauses in KB0
    for clause0 in self.KB0:
        new_clause = self.resolve(clause, clause0)
        if not new_clause:
            break
        clause = new_clause

    # Check for subsumption with all the clauses in KB
    clause_to_remove = None
    for clause2 in self.KB:
        strict_clause = self.subsumption_checking(clause, clause2)
        if not strict_clause:
            continue
        elif strict_clause == clause:
            clause_to_remove = clause2
            break
        elif strict_clause == clause2:
            return None

    if clause_to_remove:
        self.KB.remove(clause_to_remove)
    self.KB.add(clause)
    return clause

def generate_clauses(self, unmarked_neighbors, n):
    """
    About generating clauses from the hints:
    - Each hint provides the following information: There are n mines
    in a list of m unmarked cells.
    - (n == m): Insert the m single-literal positive clauses to the KB,
    one for each unmarked cell.
    - (n == 0): Insert the m single-literal negative clauses to the KB,
    one for each unmarked cell.
    - (m > n > 0): General cases (need to generate CNF clauses and add
    them to the KB):
        - C(m, m-n+1) clauses, each having m-n+1 positive literals
    """

```

```

        - C(m, n+1) clauses, each having n+1 negative literals.
        - For example, for m=5 and n=2, let the cells be x1, x2, ...,
x5:
        There are C(5,4) all-positive-literal clauses:
            (x1 or x2 or x3 or x4), (x1 or x2 or x3 or x5), ..., (x2 or
x3 or x4 or x5)
        There are C(5,3) all-negative-literal clauses:
            (not x1 or not x2 or not x3), (not x1 or not x2 or not x4),
(not x1 or not x2 or not x5), ..., (not x3 or not x4 or not x5)
        '''
        # print("generate_clauses: unmarked_neighbors len = {}, n =
{}".format(len(unmarked_neighbors), n))
        m = len(unmarked_neighbors)
        if n == m:
            for cell in unmarked_neighbors:
                self.insert_clause_to_KB(((cell[0], cell[1], True),))
        elif n == 0:
            for cell in unmarked_neighbors:
                self.insert_clause_to_KB(((cell[0], cell[1], False),))
        else:
            # Generate all-positive-literal clauses
            for combination_clause in
itertools.combinations(unmarked_neighbors, m-n+1):
                self.insert_clause_to_KB(tuple((cell[0], cell[1], True) for
cell in combination_clause))

            # Generate all-negative-literal clauses
            for combination_clause in
itertools.combinations(unmarked_neighbors, n+1):
                self.insert_clause_to_KB(tuple((cell[0], cell[1], False)
for cell in combination_clause))
        # print("finish generating clauses")

    def get_single_clause_count(self):
        '''
        Print KB, its single clause count and KB0 for debugging
        '''
        single_clause_count = 0
        for clause in self.KB:
            if len(clause) == 1:
                single_clause_count += 1
        # print(f"KB length: {len(self.KB)}, single clause count:
{single_clause_count}")
        # print(f"KB0 length: {len(self.KB0)}")

    def is_stuck(self):
        '''
        Determine if the game is stucked by KB length history
        If the KB length is the same for 5 consecutive turns, then the game
is stucked
        '''
        self.KB_length_history.append(len(self.KB))
        stuck_KB_length = 5
        if len(self.KB_length_history) > stuck_KB_length:

```

```

        self.KB_length_history.pop(0)
        if len(set(self.KB_length_history)) == 1 and
len(self.KB_length_history) == stuck_KB_length:
            return True
        return False

def make_safe_guess(self):
    """
    make a guess if the game is stucked
    choose a guess by one of the literal in the clause with the length
of 2 in KB
    if there is no clause with the length of 2, then change that to 3,
4, 5, ...
    return the guessed literal
    """
    clause_len_limit = 2
    max_clause_len = 0
    for clause in self.KB:
        if len(clause) > max_clause_len:
            max_clause_len = len(clause)
    while(clause_len_limit <= max_clause_len):
        for clause in self.KB:
            if len(clause) == clause_len_limit:
                for literal in clause:
                    if literal[2] == False:
                        print(f"make_safe_guess: {literal}")
                        # print(f"clause_len_limit: {clause_len_limit},
max_clause_len: {max_clause_len}, clause: {clause}")
                        return literal, clause

                clause_len_limit += 1
    return None, None

def game_move(self):
    """
    Perform one iteration of the game.
    - Check if the game is stucked, if so, make a safe guess
      - If the guess isn't safe, then the game is over
      - Else, mark the guess as safe and continue the game
    - If there is a single-literal clause in the KB:
      - If this cell is safe:
        - Mark that cell as safe or mined on the game board.
        - Query the game control module for the hint at that cell.
        - Generate new clauses from the hint.
      - Else:
        - Apply pairwise "matching" of the clauses in the KB or
          apply pairwise "matching" of the clauses between KB and KB0.
    """
    # If there is a single-literal clause in the KB, move it to KB0
    single_clause_count = self.get_single_clause_count()
    if self.is_stuck() and not single_clause_count:
        guess_safe_literal, guess_clause = self.make_safe_guess()
        if not guess_safe_literal:
            return None

```

```

        self.KB.remove(guess_clause)
        self.mark_safe(guess_safe_literal)
        # Query the game control module for the hint at that cell.
        unmarked_neighbors, n =
self.get_unmarked_neighbors_and_mine_count(guess_safe_literal)
        self.generate_clauses(unmarked_neighbors, n)
        self.KB0.add((guess_safe_literal,))
        return (guess_safe_literal[0], guess_safe_literal[1])

single_literal_clause = None
for clause in self.KB:
    if len(clause) == 1:
        single_literal_clause = clause
        break

if single_literal_clause:
    if not single_literal_clause[0][2]:
        self.mark_safe(single_literal_clause[0])
        # Query the game control module for the hint at that cell.
        unmarked_neighbors, n =
self.get_unmarked_neighbors_and_mine_count(single_literal_clause[0])
        self.generate_clauses(unmarked_neighbors, n)
    else:
        self.mark_mine(single_literal_clause[0])
        self.KB0.add(single_literal_clause)
        self.KB.remove(single_literal_clause)
        return (single_literal_clause[0][0], single_literal_clause[0]
[1])

# Otherwise, apply pairwise matching of clauses in the KB
# self.pairwise_matching()
self.pairwise_matching_KB0()

return None

def pairwise_matching(self):
    """
    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.
    """
    # print("pairwise_matching")
    KB = list(self.KB)

    # Iterate over all possible pairs of clauses in the knowledge base
    clauses_to_add = set()
    clauses_to_remove = set()
    for i in range(len(self.KB)):
        for j in range(i+1, len(self.KB)):
            # Check if one of the clauses has only two literals
            if len(KB[i]) <= 2 and len(KB[j]) <= 2:

```

```

        new_clause, is_strict = self.matching_clauses(KB[i],
KB[j])

        if is_strict:
            if new_clause == KB[i]:
                clauses_to_remove.add(KB[j])
            elif new_clause == KB[j]:
                clauses_to_remove.add(KB[i])
            elif new_clause:
                clauses_to_add.add(new_clause)
        for clause in clauses_to_remove:
            self.KB.remove(clause)
        for clause in clauses_to_add:
            self.insert_clause_to_KB(clause)

        # print("finish pairwise_matching")

def pairwise_matching_KB0(self):
    """
    Apply pairwise "matching" of the clauses between KB and KB0.
    If new clauses are generated due to resolution, insert them into
the KB.
    """
    # print("pairwise_matching KB0")

    # Iterate over all possible pairs between clauses in the knowledge
base and clauses in the knowledge base 0
    KB = list(self.KB)
    KB0 = list(self.KB0)
    clauses_to_add = set()
    clauses_to_remove = set()

    for i in range(len(KB0)):
        for j in range(len(KB)):
            if KB0[i] == KB[j]:
                self.KB.remove(KB[j])
                return

        new_clause, is_strict = self.matching_clauses(KB0[i],
KB[j])

        if is_strict:
            if new_clause == KB0[i]:
                clauses_to_remove.add(KB[j])
            elif new_clause:
                clauses_to_add.add(new_clause)
        for clause in clauses_to_remove:
            self.KB.remove(clause)
        for clause in clauses_to_add:
            self.insert_clause_to_KB(clause)

        # print("finish pairwise_matching KB0")

```



```
import math
import pygame
import random

# Define some colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GRAY = (128, 128, 128)
RED = (255, 0, 0)
BLUE = (0, 0, 255)

CELL_SIZE = 40

class MinesweeperGame():
    def __init__(self, rows, cols, mines, initial_safe_cells_times):
        self.rows = rows
        self.cols = cols
        self.mines = mines
        self.width = self.cols * CELL_SIZE
        self.height = self.rows * CELL_SIZE
        self.board = []
        self.initial_safe_cells_times = initial_safe_cells_times
        self.create_board()
        self.generate_mine()
        self.screen = self.initialize_pygame()
        self.initial_random_safes = self.get_initial_safe_cells()

    def initialize_pygame(self):
        # Initialize Pygame
        pygame.init()
        pygame.display.set_caption("Minesweeper")
        screen = pygame.display.set_mode((self.width, self.height))
        return screen

    def create_board(self):
        # Create the game board
        for row in range(self.rows):
            self.board.append([])
            for col in range(self.cols):
                self.board[row].append({'mine': False, 'revealed': False,
'marked': False})

    def generate_mine(self):
        # Add some random mines
        mine_count = 0
        while mine_count < self.mines:
            row = random.randint(0, self.rows-1)
            col = random.randint(0, self.cols-1)
            if not self.board[row][col]['mine']:
                self.board[row][col]['mine'] = True
                mine_count += 1
```

```

def get_initial_safe_cells(self):
    num_initial_safe_cells = round(math.sqrt(self.rows * self.cols)) *
self.initial_safe_cells_times
    safe_cells = set()
    while len(safe_cells) < num_initial_safe_cells:
        row = random.randint(0, self.rows - 1)
        col = random.randint(0, self.cols - 1)
        if not self.board[row][col]['mine'] and (row, col) not in
safe_cells:
            safe_cells.add((row, col))
    return safe_cells

def get_adjacent_mines(self, row, col):
    count = 0
    for r in range(max(row-1, 0), min(row+2, self.rows)):
        for c in range(max(col-1, 0), min(col+2, self.cols)):
            if self.board[r][c]['mine']:
                count += 1
    return count

def draw_board(self, win_or_lose=None):
    self.screen.fill(GRAY)
    for row in range(self.rows):
        for col in range(self.cols):
            x = col * CELL_SIZE
            y = row * CELL_SIZE
            cell = self.board[row][col]
            if not cell['revealed']:
                color = WHITE
                if cell['marked']:
                    pygame.draw.rect(self.screen, RED, (x, y,
CELL_SIZE, CELL_SIZE))
                else:
                    color = BLACK
                    if cell['mine']:
                        pygame.draw.circle(self.screen, BLACK,
(x+CELL_SIZE//2, y+CELL_SIZE//2), CELL_SIZE//4)
                    else:
                        count = self.get_adjacent_mines(row, col)
                        if count >= 0:
                            font = pygame.font.SysFont(None, CELL_SIZE//2)
                            text = font.render(str(count), True, BLACK)
                            self.screen.blit(text, (x+CELL_SIZE//4,
y+CELL_SIZE//4))
                        pygame.draw.rect(self.screen, color, (x, y, CELL_SIZE,
CELL_SIZE), 1)
            if win_or_lose is not None:
                font = pygame.font.SysFont(None, 50)
                text = font.render(win_or_lose, True, BLUE)
                self.screen.blit(text, (self.width/2 - text.get_width()/2,
self.height/2 - text.get_height()/2))

# draw board
# X is for mine == True

```

```

# U is for revealed == False
# M is for marked == True
# 0-8 is for mines number from get_adjacent_mines(row, col)
def draw_board_debug(self):
    for row in range(self.rows):
        for col in range(self.cols):
            cell = self.board[row][col]
            if cell['mine']:
                print('X', end=' ')
            elif cell['revealed']:
                print(self.get_adjacent_mines(row, col), end=' ')
            elif cell['marked']:
                print('M', end=' ')
            else:
                print('U', end=' ')
        print()

def mark_safe(self, row, col):
    cell = self.board[row][col]
    if not cell['revealed']:
        cell['revealed'] = True

def mark_mine(self, row, col):
    cell = self.board[row][col]
    if not cell['marked']:
        cell['marked'] = True

def get_unmarked_neighbors_and_mine_count(self, row, col):
    neighbors = []
    mine_count = 0
    for r in range(max(row-1, 0), min(row+2, self.rows)):
        for c in range(max(col-1, 0), min(col+2, self.cols)):
            if (r != row or c != col):
                if not self.board[r][c]['revealed'] and not
self.board[r][c]['marked']:
                    if self.board[r][c]['mine']:
                        mine_count += 1
                    neighbors.append((r, c))
    return neighbors, mine_count

def lose(self):
    for row in range(self.rows):
        for col in range(self.cols):
            cell = self.board[row][col]
            if cell['revealed'] and cell['mine']:
                return True
    return False

def win(self):
    marked_mines = 0
    for row in range(self.rows):
        for col in range(self.cols):
            cell = self.board[row][col]
            if cell['marked'] and cell['mine']:

```

```

        marked_mines += 1
        if not cell['revealed'] and not cell['marked']:
            return False
    if marked_mines != self.mines:
        return False
    return True

```

### RunGame.py:

```

import sys
import pygame
from MineSweeperGame import MineSweeperGame
from MineSweeperAI import MineSweeperPlayer

difficulty = 'easy'
initial_safe_cells_times = 1

if len(sys.argv) > 1:
    difficulty = sys.argv[1]
if len(sys.argv) > 2:
    initial_safe_cells_times = int(sys.argv[2])

if difficulty.lower() == "easy":
    game = MineSweeperGame(rows=9, cols=9, mines=10,
initial_safe_cells_times=initial_safe_cells_times)
elif difficulty.lower() == "medium":
    game = MineSweeperGame(rows=16, cols=16, mines=25,
initial_safe_cells_times=initial_safe_cells_times)
elif difficulty.lower() == "hard":
    game = MineSweeperGame(rows=16, cols=30, mines=99,
initial_safe_cells_times=initial_safe_cells_times)

player = MineSweeperPlayer(game)

# print game board
# game.draw_board_debug()

# Game loop
running = True
first_click = True
step = 0
win_or_lose = None
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            # Exit the game loop when the window is closed
            running = False

    if not win_or_lose:
        if game.win():
            win_or_lose = "You win!"
        elif game.lose():

```

```
        win_or_lose = "You lose!"

    if not win_or_lose:
        step += 1
        move = player.game_move()
        # print(f"step: {step}, move: {move}")

    # pygame.time.wait(100)
    # Draw the board
    game.draw_board(win_or_lose)
    # Update the screen
    pygame.display.flip()
# Quit Pygame
pygame.quit()
```