Report

109550005 張可晴

1. INTRODUCTION

In this project, I implement Minesweeper game by referencing the template provided by Harvard CS50AI. There are two files in this project. One is minesweeper.py, which is mainly responsible for developing the propositional logical agent and game control modules. The other is runner.py, which handles the GUI based on the package from pygame. In addition, minesweeper.py also visualization the game process by showing it on the terminal if users want this function.

2. IMPLEMENTATION

In this part of the report, I will introduce the detailed implementation of this project. I will cover all the requirements listed in the spec in this part.

2.1 GAME CONTROL MODULE

In my case, I will provide 3 different board sizes and the number of mines initially. By referencing the spec, there are 3 levels for the user to choose at the beginning of the games, which is Easy (9x9 board with 10 mines), Medium (16x16 board with 25 mines), and Hard (30x16 board with 99 mines). The following figure is the implementation that users can pick up which game level they want to play at the very beginning of the game.

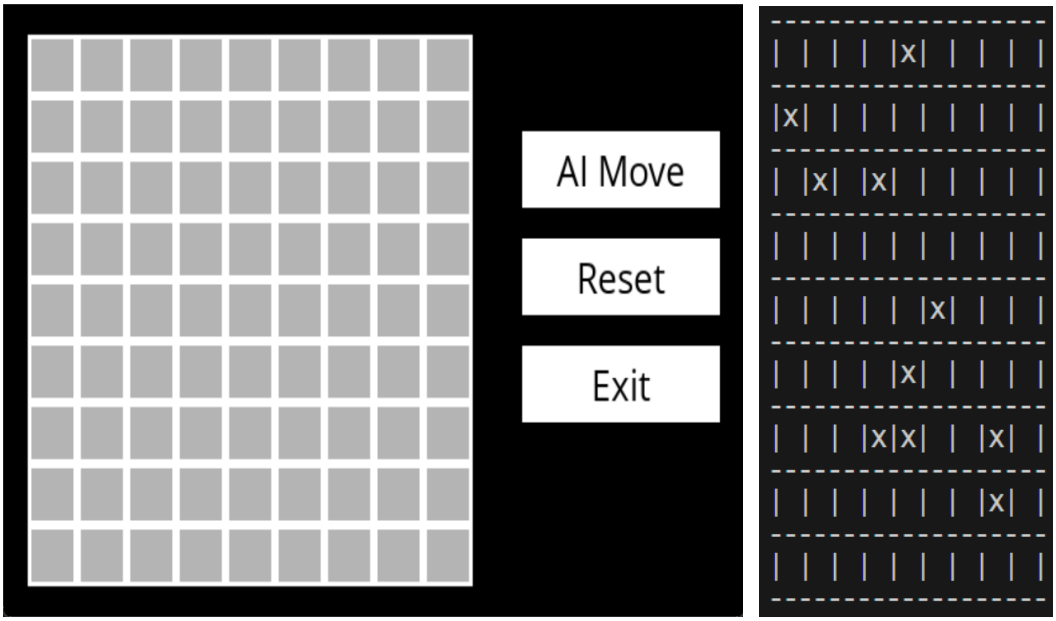
```
PS C:\Users\cassie\Desktop\Artificial-Intelligence-Capstone\project3> python minesweeper.py -t 1 -s 1
pygame 2.3.0 (SDL 2.24.2, Python 3.7.9)
Hello from the pygame community. https://www.pygame.org/contribute.html
Please chose the difficulty:
(A) Easy (9x9 board with 10 mines)
(B) Medium (16x16 board with 25 mines)
(C) Hard (30x16 board with 99 mines)
```

As for providing the hints when queried for a cell by the player module, I will go through 9 cells whose center is just the queried cell, and skip the query one and the illegal ones. In the end, if the symbol of the cell is True, which suggests that there is a mine, then I will plus one for the keeper that keeps the total number of surrounding mines.

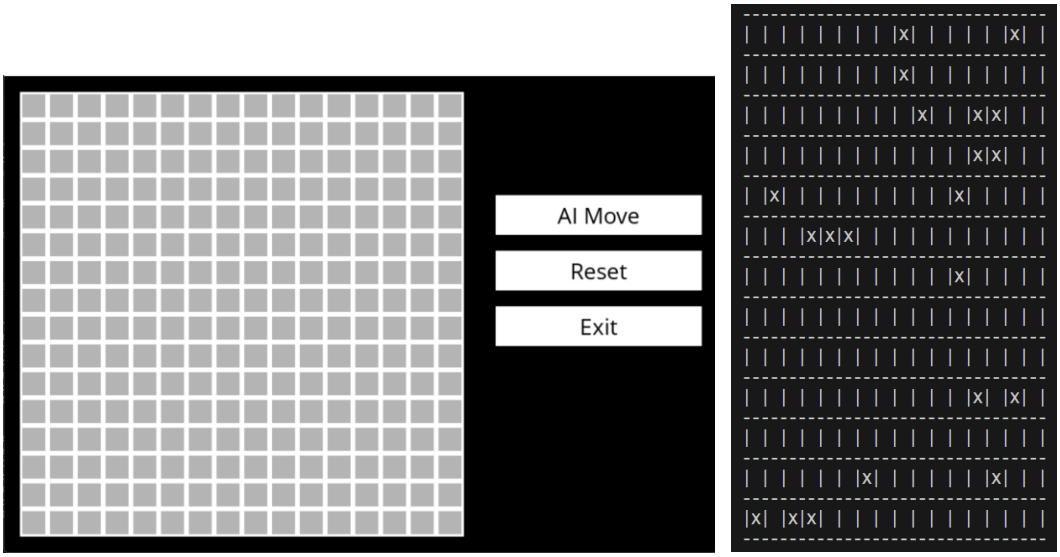
In terms of the initial list of safe cells, in this assignment, we are asked for a randomly preset s list of safe cells so that the agents do not need to randomly pick up a grid at the beginning of the game. Therefore, I first construct a list to keep the

initial safe cells. After that, I will use random.randrange function to randomly choose the position in a row and column, then check if the symbol of that chosen cell is False. If it is False, which means there is no mine, then add it to the list. Otherwise, skip that cell. The procedure will keep going until the number of collected safe cells meets our expectations. To begin with, I use round(sqrt(#cells)) as the number of initial safe cells, while I will tune this parameter and discuss the effect on the win rate later.

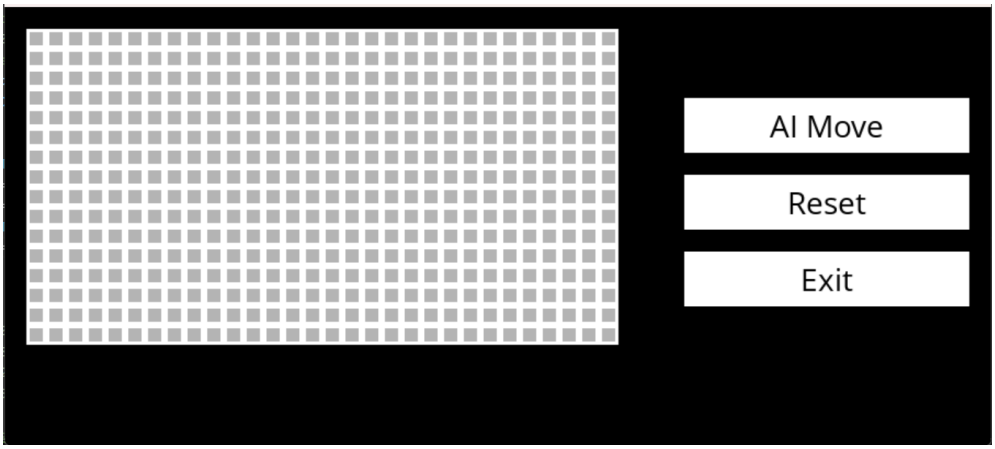
The followings are the GUI of 3 different levels of Minesweeper at the initial phase.

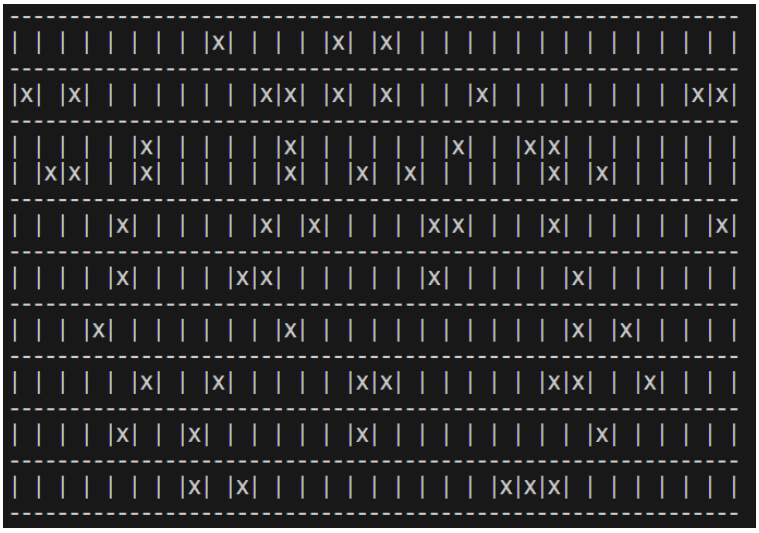


Easy (9x9 board with 10 mines)



Medium (16x16 board with 25 mines)

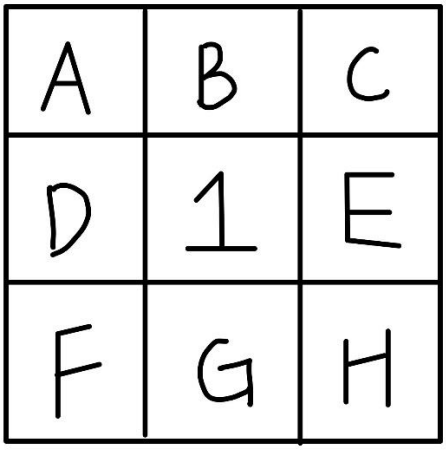




Hard (30x16 board with 99 mines)

2.2 REPRESENTATION

For implementation, I don't precisely use the CNF clauses as the representation of each cell on the game board. Instead, I will extend the concept and take advantage of the alternative representation method. The reason for utilizing other representation methods is to reduce space complexity. Take the following scenario as an example.



If we use the traditional CNF clauses to represent this case, it will need to use 8 clauses to express this situation. That is,

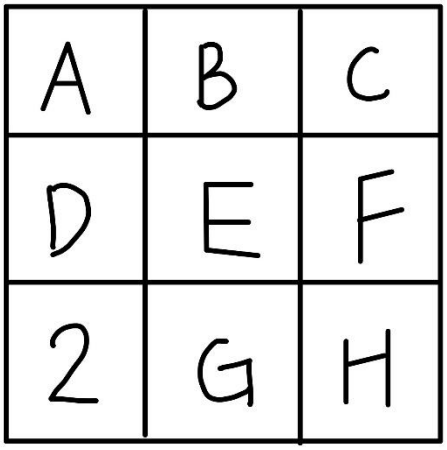
(A ∨ not B ∨ not C ∨ not D ∨ not E ∨ not F ∨ not G ∨ not H) ∧
(not A ∨ B ∨ not C ∨ not D ∨ not E ∨ not F ∨ not G ∨ not H) ∧
(not A ∨ not B ∨ C ∨ not D ∨ not E ∨ not F ∨ not G ∨ not H) ∧
(not A ∨ not B ∨ not C ∨ D ∨ not E ∨ not F ∨ not G ∨ not H) ∧
(not A ∨ not B ∨ not C ∨ not D ∨ E ∨ not F ∨ not G ∨ not H) ∧
(not A ∨ not B ∨ not C ∨ not D ∨ not E ∨ F ∨ not G ∨ not H) ∧
(not A ∨ not B ∨ not C ∨ not D ∨ not E ∨ not F ∨ G ∨ not H) ∧
(not A ∨ not B ∨ not C ∨ not D ∨ not E ∨ not F ∨ not G ∨ H)

What's worse, it is just the case when the hint is 1. If the hint is 2 or 3, there are even much more expressions needed to be stored. Also, if the board size is 9x9 grids, which means that we have 81 cells, then we need to check for 2 ^ 81 possible cases. It is quite a huge amount to compute. Not to mention that it is merely the Easy one, there are Medium and Difficulty for users to play.

To solve this problem, I take a more intuitive and easy-implemented way to represent the knowledge sentence. In my assignment, I will rather use the following way to represent the same scenario.

{A, B, C, D, E, F, G, H} = 1

The above logical sentence indicates that there is 1 mine among A, B, C, D, E, F, G, and H. On this basis, take the following case as example.



The representation will be:

$$\{D, E, G\} = 2$$

It reveals that there are 2 mines among D, E, and F. As for the implementation, I design a Class *Sentence* to realize this concept. It consists of 2 parts. One is a set that keeps the coordinate of the cell, while the other is a counter keeping the number of the surrounding mines.

2.3 Player Module

To begin with, I prepare *KB*, which is a list to keep the logical sentence, and *KB0*, which is a set to keep the cells revealed by the player. Besides, to prevent the KB from being too large, I additionally construct 2 sets, *safes,* and *mines*. They are employed to keep the cells, which is sure to be safe cell or mine, respectively.

Moreover, when the game starts, the initial safe cells will be all put in the *KB* and *safes*, which suggests that we are sure that those cells are all safe. On the other hand, *KB0* is empty as the spec required.

2.4 Game Flow

There is a little difference between my game flow and the one listed on the spec, while most of the parts are the same. The following is my game flow:

The game play proceeds in a loop. Within each iteration:

If there is a cell in the *safes* or *mines* and not in the *KB0*:

Mark that cell as safe or mined.

Move that clause to *KB0*.

Process the "matching" of that clause to all the remaining clauses in the

KB.

If new clauses are generated due to resolution, insert them into the *KB* or *safes* or *mines*.

If this cell is safe:

Query the game control module for the hint at that cell.

Insert the clauses regarding its unmarked neighbors into the *KB*.

Otherwise:

Apply pairwise "matching" of the clauses in the *KB*.

If new clauses are generated due to resolution, insert them into the *KB* or *safes* or *mines*.

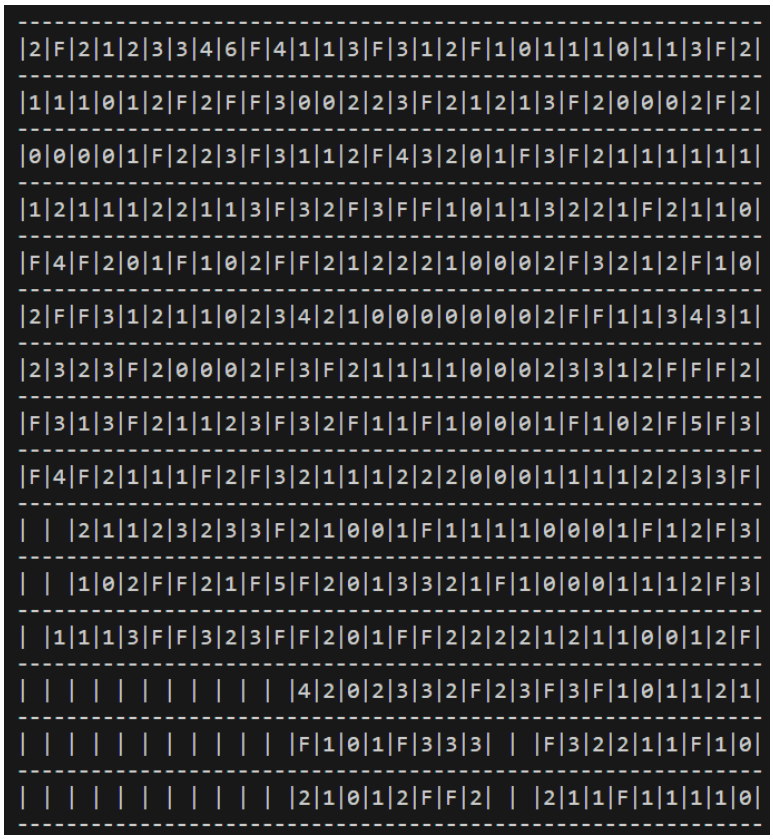
In a word, because there are additional *safes* and *mines*, something will put into KB such as a single-lateral clause will no longer do so. By contrast, it will be directly put into *safes* or *mines*. In this way, there are two advantages. On the one hand, we do not need to wait for checking *KB* whether there is a single-lateral clause when making a move. We can just pick up one of the unvisited moves from *safes* or *mines.* On the other hand, KB will not take too much memory thanks to the *safes* and *mines*. In this way, we do not need to be concerned about whether KB will grow too fast.

2.5 GAME TERMINATION

In each turn of Minesweeper, I will construct two lists, flags and marked. Flags is for storing the cells considered to be mine, while marked stores the cells regarded as safe ones. For success, if the sum of the size of flags and the size of marked is equal to the board size. The game is over and the propositional logical agent wins.

As for lose, once the player makes a move that the player thinks is safe, while it is mine in fact, the player loses the game.

Last but not least, in real game rules, Minesweeper allows the player randomly place a piece when there is no new cell inferred by logic. Nevertheless, in this assignment, we just need to announce the game is stuck and let it over, when there is no new generated cell. To be more specific, it is commonplace for the Hard level when the game is near done. Additionally, In my assignment, If the player does not have the new cell to mark in 5 consecutive turns, the game will be determined stuck. The following is an example of a “stuck” game when the game level is Hard.

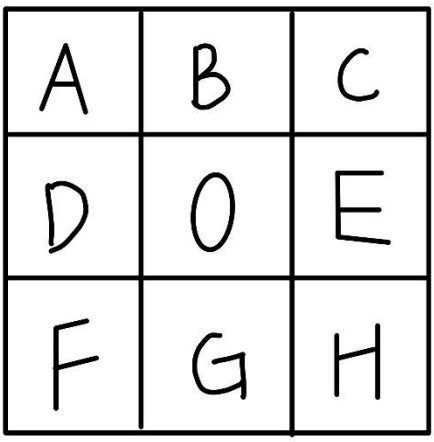


2.6 Generating Clauses From the Hints

In this part, since my representation method of logical statement is different from the CNF clauses, although the concept is the same, I do not need to take as many as the spec mentioned conditions into consideration. In contrast, I initially need to apply the function mentioned in the game control module to query the number of nearby mines. After that, I will go through all the surrounding cells to find the unmarked cell and put it into a set. To elaborate, I will pass the cells in *safes*, *mine*, or *KB0*. What's more, if the cell in *mines*, I will reduce the number of surrounding mine just queried by 1. Finally, I will construct a new sentence composed of the set mentioned above and the final count of nearby mines.

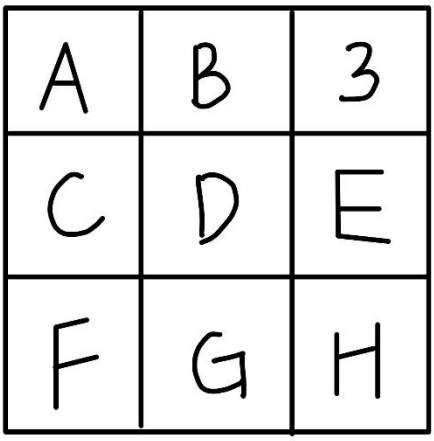
2.7 Inserting a New Clause to the KB

First of all, I will do a resolution of the new sentence with all the cells in *safes* and *mine*, if applicable. There are 2 cases triggering the certain inference. The following are the explanations with figures.



{A, B, C, D, E, F, G, H} = 0

For the above case, it is intuitive for us to infer that A, B, C, D, E, F, G, and H are all safe since there is no mine among them.



{B, D, E} = 3

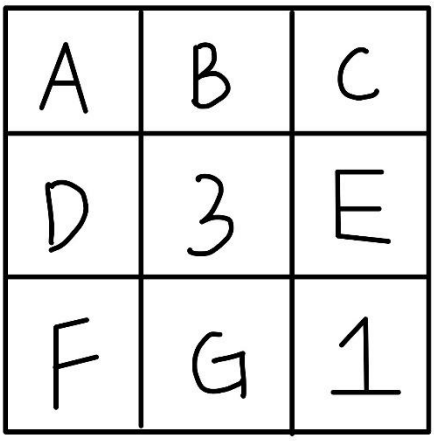
It is also obvious for us to infer that B, D, and E are all the mines because there are 3 mines among 3 of them. In conclusion, if the number of nearby mines of the new sentence is 0, all the cells in the sentence are safe. If the number of surrounding mines is the same as the number of cells in the sentence, all the cells in the sentence are mines. Moreover, once one of the above 2 cases occurs, I will do unit-propagation by adding the cells into *safes* or *mines*, remove those cells from all the sentences in the *KB*, and reduce the number of nearby mines kept in the sentences by 1 if the cell is mine.

Secondly, If the new sentence is identical to one of the sentences in the *KB,* it cannot be inserted into the *KB.*

Finally, since my representation will not bump into subsumption, I do not need to handle such a case. The sentence in the *KB* will always be the most strict one.

2.8 MATCHING

As for this part, I will check the duplication first. If there are identical sentences, then discard one of them. For subsumption, I will only keep the more strict sentence in the *KB.* As a result, If one of the sets keeping the surrounding cells from one of the sentences in the *KB* is the subset of one of the other sets of the sentence in the *KB.* I will let the subset keep staying in the *KB,* while I will remove the superset and reinsert the other part of the superset into the *KB.* The following will provide a more detailed explanation with the figure.



In this case, this board state will generate 2 sentences listed below in the begining.

{A, B, C, D, E, F, G} = 3
{E, G} = 1

Then, we find that the first one is the superset of the second one. Then, we can make an inference from these 2 sentences and it can be transformed into the below representation:

{A, B, C, D, F} = 2
{E, G} = 1

Therefore, we will remove {A, B, C, D, E, F, G} = 3 from the *KB*, then inserting {A, B, C, D, F} = 2 into the *KB*.

Additionally, I will also do the resolution and unit-propagation mentioned in the last part.

3. Experiments

I have conducted 2 experiments. For the first one, I let each agent with the different game level play 100 turns when the number of the first initial safe cell is all equal to round(sqrt(#cells)), then compare their win rate. From the below table, we can find that it is worth noticing that the agent with the game level Easy and Medium are well-matched in strength, while the Hard level agent performs much worse than them. Accordingly, I try to fine-tune the number of the initial safe cell to observe its impact on the win rate of the Hard level agent.

LEVEL	WIN RATE (%)
Easy	96
Medium	97
Hard	25

The second experiment aims to observe the effect of the number of initial safe cells on the win rate when the game level is Hard. From the below table, it is no surprise that the more the number of initial safe cells is, the higher the win rate is. It makes sense because all the agents that do not win the game are all because the game gets stuck. Hence, if there is more initial safe cells, the opportunity that the game is stuck will become less.

# OF INITIAL SAFE CELL	WIN RATE (%)
22	25
53	40
120	50

4. Challenge

For me, the most challenging part is absolutely the representation of the logical sentence. I initially use the CNF clauses provided by PyPI, but I had a hard time implementing them. Thanks to the representation proposed by Harvard CS50AI, I can finish this project.

5. References

https://cs50.harvard.edu/ai/2020/projects/1/minesweeper/

APPENDIX

MINESWEEPER.PY

```python
import pygame
import sys
import time
import itertools
import random
import math
import argparse

# false -> safe
# true -> mine

# TODO: changing this can apply to different board size
# e.g. 9 * 9 -> 10 mines, 16 * 16 -> 25 mines, 30 * 16 -> 99 mines
HEIGHT = 9
WIDTH = 9
MINES = 10

def get_parser():
    parser = argparse.ArgumentParser(description="the number of turn")
    parser.add_argument("-t", "--turn", default=1, type=int)
    parser.add_argument("-s", "--show", default=0, type=int)

    return parser.parse_args()

class GameControl():
    """
    Minesweeper game representation
    GAME CONTROL
    """
```

```python
    def __init__(self, height=HEIGHT, width=WIDTH, mines=MINES):
        # Set initial width, height, and number of mines
        self.height = height
        self.width = width
        self.mines = set()

        # Initialize an empty field with no mines
        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(height)
            j = random.randrange(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()

        # initial safe list
        # TODO change the number of initial safe list to observer the win rate
        self.initial_safes = []
        while len(self.initial_safes) != int(round(math.sqrt(self.width * self.height), 0)):
```

```python
                i = random.randrange(height)
                j = random.randrange(width)
                if not self.board[i][j]:
                    self.initial_safes.append((i, j))

    def print(self):
        """
        Prints a text-based representation
        of where mines are located.
        """
        for i in range(self.height):
            print("--" * self.width + "-")
            for j in range(self.width):
                if self.board[i][j]:
                    print("|X", end="")
                else:
                    print("| ", end="")
            print("|")
        print("--" * self.width + "-")

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

    def hint(self, cell):
        """
        Provide the hint (number of surrounding mines)
        """
        # for keep the number of surrounding mines
```

```
        count = 0
        for i in range(cell[0] - 1, cell[0] + 2):
            # if the position is illegal, then skip
            if i < 0 or i >= self.height:
                continue
            for j in range(cell[1] - 1, cell[1] + 2):
                # if the position is illegal or just the same with cell, then skip
                if j < 0 or j >= self.width or (i == cell[0] and j == cell[1]):
                    continue

                # the position has mine, so count++
                if self.board[i][j]:
                    count += 1

        return count

    def get_initial_safes(self):
        """
        get the initial safe list
        """
        return self.initial_safes

    def print_current(self, marked, flags):
        """
        Prints a text-based representation
        of where current board state.
        """
        for i in range(HEIGHT):
            print("--" * WIDTH + "-")
```

```
            for j in range(WIDTH):
                if (i, j) in flags:
                    print("|F", end="")
                elif (i, j) in marked:
                    text = "|%s" % str(self.hint((i, j)))
                    print(text, end="")
                else:
                    print("| ", end="")
            print("|")
        print("--" * WIDTH + "-")


class Sentence():
    """
    Logical statement about a Minesweeper game
    A sentence consists of a set of board cells,
    and a count of the number of those cells which are mines.
    """

    def __init__(self, cells, count):
        # keeping the coordinate of the cells
        self.cells = set(cells)
        # keep the number of the surrouding mines
        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):
        """
        Returns the inferred mines
        """
        res = set()
        # when the number of surrounding mins is equal to the surrounding unknown mines,
        # it means that all of them are mines
        if len(self.cells) == self.count and self.count != 0:
            res = self.cells

        return res

    def known_safes(self):
        """
        Returns the inferred safe cells
        """
        res = set()
        # when the surrounding mines is 0,
        # it means that all the surrouding cells are safe
        if self.count == 0:
            res = self.cells

        return res

    def mark_mine(self, cell):
        """
        update the information to the sentence that there is a mine for sure
        """
        # if the mine is in the sentence,
```

```
        # then move it out and reduce the count by one
        if cell in self.cells:
            self.cells.remove(cell)
            self.count -= 1

    def mark_safe(self, cell):
        """
        update the information to the sentence that there is a safe cell for sure
        """
        # if the mine is in the sentence,
        # then move it out and no need to reduce the count
        # since it is count for the number of mines
        if cell in self.cells:
            self.cells.remove(cell)


class Player():
    """
    Minesweeper game player
    PLAYER
    """
    def __init__(self, game, height=HEIGHT, width=WIDTH):
        # TODO change size
        # Set initial height and width
        self.height = height
        self.width = width

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

```
        # List of sentences about the game known to be true
        # KB(sentence) -> list(set())
        self.KB = []
        # self.KB = game.get_initial_safes()
        # print(self.KB[0])
        for cell in game.get_initial_safes():
            self.safes.add(cell)
            cells = set()
            cells.add(cell)
            self.KB.append(Sentence(cells, 0))

        # represent marked cell
        # set(cell) -> set(set())
        # TODO change to list()
        self.KB0 = set()

    def mark_mine(self, cell):
        """
        UNIT PROPAGATION for mines
        """
        self.mines.add(cell)
        for sentence in self.KB:
            sentence.mark_mine(cell)

    def mark_safe(self, cell):
        """
        UNIT PROPAGATION for safe cell
        """
        self.safes.add(cell)
```

```
        for sentence in self.KB:
            sentence.mark_safe(cell)

    def make_move(self, toPrint):
        """
        Mark that cell as safe or mine, move that clause to KB0.
        """
        # the returned move
        moves = set()
        if toPrint:
            print('current KB:')
            for sentence in self.KB:
                print(sentence.cells, end=' ')
                print(sentence.count)

        for safe in self.safes:
            if safe not in self.KB0:
                # Move that calsue to KB0
                self.KB0.add(safe)
                self.mark_safe(safe)
                moves.add(safe)
                break

        if len(moves) != 0:
            return moves.pop()
        else:
            for mine in self.mines:
                if mine not in self.KB0:
                    # Move that calsue to KB0
```

```
                    self.KB0.add(mine)
                    self.mark_mine(mine)
                    moves.add(mine)
                    break

            if len(moves) == 0:
                return None
            else:
                return moves.pop()

    def insert_new_clause(self, cell, count):
        """
            1. generating new sentencce from the hints
            2. insert the clauses regarding its unmarked neighbors into the KB
        """
        unmarked_neibors = set()

        # 1. generating new sentencce from the hints
        for i in range(cell[0] - 1, cell[0] + 2):
            if i < 0 or i >= self.height:
                continue
            for j in range(cell[1] - 1, cell[1] + 2):
                if j < 0 or j >= self.width or (i == cell[0] and j == cell[1]):
                    continue

                # skip if marked
                if (i, j) in self.safes or (i, j) in self.mines or (i, j) in self.KB0:
                    if (i, j) in self.mines:
                        count -= 1
                    continue
```

```
                unmarked_neibors.add((i, j))

        new_sentence = Sentence(unmarked_neibors, count)
        safes = new_sentence.known_safes()
        mines = new_sentence.known_mines()

        # do resolution and insert the new sentence in KB
        if safes != set():
            keep = safes.copy()
            for safe in keep:
                self.mark_safe(safe)

        elif mines != set():
            keep = mines.copy()
            for mine in keep:
                self.mark_mine(mine)

        # Skip the insertion if there is an identical clause in KB.
        elif new_sentence in self.KB:
            pass

        else:
            self.KB.append(new_sentence)

    def matching(self):
        """
            1. duplication
            2. matching
            If new clauses are generated due to resolution, insert them into the KB or safes or mins
        """
```

```
        # 1. duplication
        # keep the sentence needed to be removed owning to duplication and resolution
        removed_sentence = []
        # keep the inferred sentence needed to be appended owning to resolution
        stricker_sentence = []

        if len(self.KB) != 0:
            for i in range(len(self.KB)):
                for j in range(len(self.KB)):
                    if i >= j:
                        continue

                    sentence1 = self.KB[i]
                    sentence2 = self.KB[j]

                    # duplication
                    if sentence1 == sentence2 and sentence1.cells != set():
                        removed_sentence.append(sentence2)

                    # resolution
                    elif sentence1.cells.issubset(sentence2.cells):
                        stricker_sentence.append(Sentence(sentence2.cells - sentence1.cells, sentence2.count - sentence1.count))
                        removed_sentence.append(sentence2)

                    elif sentence1.cells.issuperset(sentence2.cells):
                        stricker_sentence.append(Sentence(sentence1.cells - sentence2.cells, sentence1.count - sentence2.count))
                        removed_sentence.append(sentence1)

        if len(removed_sentence) != 0:
```

```
            for sentence in removed_sentence:
                if sentence in self.KB:
                    self.KB.remove(sentence)

        if len(stricker_sentence) != 0:
            for sentence in stricker_sentence:
                if sentence not in self.KB:
                    self.KB.append(sentence)

        # 2. matching
        for sentence in self.KB:
            # check for safe cell
            safes = sentence.known_safes()
            if safes != set():
                keep = safes.copy()
                for safe in keep:
                    self.mark_safe(safe)

            mines = sentence.known_mines()
            if mines != set():
                keep = mines.copy()
                for mine in keep:
                    self.mark_mine(mine)


if __name__ == '__main__':
    parser = get_parser()
    turn = parser.turn
    toPrint = parser.show
    win = 0

    # ask the player for the game level
    ans = input('Please chose the difficulty:\n(A) Easy (9x9 board with 10 mines)\n\
(B) Medium (16x16 board with 25 mines)\n\
(C) Hard (30x16 board with 99 mines)\n')

    if ans == 'B' or ans == '(B)' or ans == 'b':
        HEIGHT, WIDTH, MINES = 16, 16, 25
    elif ans == 'C' or ans == '(C)' or ans == 'c':
        HEIGHT, WIDTH, MINES = 16, 30, 99

    for i in range(turn):
        # Create game and AI agent
        game = GameControl(height=HEIGHT, width=WIDTH, mines=MINES)
        ai = Player(game, height=HEIGHT, width=WIDTH)

        # Keep track of revealed cells, flagged cells, and if a mine was hit
        marked = set()
        flags = set()
        lost = False

        # Show instructions initially
        instructions = True
        initial = True
        count = 1
        stuck = 0

        while 1:
            # show welcome
            if instructions and toPrint:
                print('****************************************WELCOME TO MINESWEEPER****************************************')
                print('self initial safe list:')
                for safe in game.initial_safes:
                    print(safe, end=' ')
                print('mine:')
                print(game.mines)
                initial = False
                game.print()
                instructions = False

            # win or lose or not yet
            if lost:
                print('Game #%s LOSE' % str(i + 1))
                break
            elif len(flags) + len(marked) == HEIGHT * WIDTH:
                print('Game #%s WIN' % str(i + 1))
                win += 1
                break
            elif stuck == 5:
                print('Game #%s STUCK' % str(i + 1))
                print(len(flags))
                break
            elif toPrint:
                print('Turn #%d' % count)
                count += 1

            # start game flow
            move = ai.make_move(toPrint)
```

```
            # move = ai.make_safe_move()
            if toPrint:
                print('move:', end=' ')
                print(move)

            if move:
                # Process the "matching" of that clause to all the remaining clauses in the KB.
                ai.matching()
                stuck = 0
                if move in ai.mines:
                    flags.add(move)
                elif game.is_mine(move):
                    lost = True
                else:
                    # this cell is safe
                    # Query the game control module for the hint at that cell
                    hint = game.hint(move)
                    ai.insert_new_clause(move, hint)
                    # keep for visulaizaton
                    marked.add(move)
            else:
                ai.matching()
                stuck += 1

            # print current board state
            if toPrint:
                game.print_current(marked, flags)

        # for checking use
        if toPrint:
```

```
            game.print()

    win /= turn
    print('\nWIN RATE: %f' % win)
```

RUNNER.PY

```
import pygame
import sys
import time

from minesweeper import GameControl, Player

ans = input('Please chose the difficulty:\n(A) Easy (9x9 board with 10 mines)\n\
(B) Medium (16x16 board with 25 mines)\n\
(C) Hard (30x16 board with 99 mines)\n')

# default (A)
# number of grid
HEIGHT = 9
WIDTH = 9
# number of mines
MINES = 10
# window size
size = width, height = 600, 400
BOARD_PADDING = 20

if ans == 'B' or ans == '(B)' or ans == 'b':
    HEIGHT, WIDTH, MINES = 16, 16, 25
    size = width, height = 900, 600
elif ans == 'C' or ans == '(C)' or ans == 'c':
    HEIGHT, WIDTH, MINES = 16, 30, 99
    size = width, height = 900, 400


# Colors
```

```
BLACK = (0, 0, 0)
GRAY = (180, 180, 180)
WHITE = (255, 255, 255)
RED = (255, 0, 0)

# Create game
pygame.init()
screen = pygame.display.set_mode(size)

# Fonts
OPEN_SANS = "assets/fonts/OpenSans-Regular.ttf"
smallFont = pygame.font.Font(OPEN_SANS, 20)
mediumFont = pygame.font.Font(OPEN_SANS, 28)
largeFont = pygame.font.Font(OPEN_SANS, 40)

# Compute board size
board_width = ((2 / 3) * width) - (BOARD_PADDING * 2)
board_height = height - (BOARD_PADDING * 2)
cell_size = int(min(board_width / WIDTH, board_height / HEIGHT))
board_origin = (BOARD_PADDING, BOARD_PADDING)

# Add images
flag = pygame.image.load("assets/images/flag.png")
flag = pygame.transform.scale(flag, (cell_size, cell_size))
mine = pygame.image.load("assets/images/mine.png")
mine = pygame.transform.scale(mine, (cell_size, cell_size))

# Create game and AI agent
game = GameControl(height=HEIGHT, width=WIDTH, mines=MINES)
```

```
ai = Player(game, height=HEIGHT, width=WIDTH)

# Keep track of revealed cells, flagged cells, and if a mine was hit
marked = set()
flags = set()
lost = False

# Show instructions initially
instructions = True
first_move = True

while True:

    # Check if game quit
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    screen.fill(BLACK)

    # Show game instructions
    if instructions:

        # Title
        title = largeFont.render("Play Minesweeper", True, WHITE)
        titleRect = title.get_rect()
        titleRect.center = ((width / 2), 50)
        screen.blit(title, titleRect)
```

```
        # Rules
        rules = [
            " ",
            "Click a cell to reveal it.",
            "Right-click a cell to mark it as a mine.",
            "Mark all mines successfully to win!"
        ]
        for i, rule in enumerate(rules):
            line = smallFont.render(rule, True, WHITE)
            lineRect = line.get_rect()
            lineRect.center = ((width / 2), 150 + 30 * i)
            screen.blit(line, lineRect)

        # Play game button
        buttonRect = pygame.Rect((width / 4), (3 / 4) * height, width / 2, 50)
        buttonText = mediumFont.render("Play Game", True, BLACK)
        buttonTextRect = buttonText.get_rect()
        buttonTextRect.center = buttonRect.center
        pygame.draw.rect(screen, WHITE, buttonRect)
        screen.blit(buttonText, buttonTextRect)

        # Check if play button clicked
        click, _, _ = pygame.mouse.get_pressed()
        if click == 1:
            mouse = pygame.mouse.get_pos()
            if buttonRect.collidepoint(mouse):
                instructions = False
                time.sleep(0.3)
```

```
            pygame.display.flip()
            continue

        # Draw board
        cells = []
        for i in range(HEIGHT):
            row = []
            for j in range(WIDTH):

                # Draw rectangle for cell
                rect = pygame.Rect(
                    board_origin[0] + j * cell_size,
                    board_origin[1] + i * cell_size,
                    cell_size, cell_size
                )
                pygame.draw.rect(screen, GRAY, rect)
                pygame.draw.rect(screen, WHITE, rect, 3)

                # Add a mine, flag, or number if needed
                if game.is_mine((i, j)) and lost:
                    screen.blit(mine, rect)
                elif (i, j) in flags:
                    screen.blit(flag, rect)
                elif (i, j) in marked:
                    neighbors = smallFont.render(
                        str(game.hint((i, j))),
                        True, BLACK
                    )
                    neighborsTextRect = neighbors.get_rect()
```

```
                    neighborsTextRect.center = rect.center
                    screen.blit(neighbors, neighborsTextRect)

                row.append(rect)
            cells.append(row)

        # AI Move button
        aiButton = pygame.Rect(
            (2 / 3) * width + BOARD_PADDING, (1 / 3) * height - 50,
            (width / 3) - BOARD_PADDING * 2, 50
        )
        buttonText = mediumFont.render("AI Move", True, BLACK)
        buttonRect = buttonText.get_rect()
        buttonRect.center = aiButton.center
        pygame.draw.rect(screen, WHITE, aiButton)
        screen.blit(buttonText, buttonRect)

        # Reset button
        resetButton = pygame.Rect(
            (2 / 3) * width + BOARD_PADDING, (1 / 3) * height + 20,
            (width / 3) - BOARD_PADDING * 2, 50
        )
        buttonText = mediumFont.render("Reset", True, BLACK)
        buttonRect = buttonText.get_rect()
        buttonRect.center = resetButton.center
        pygame.draw.rect(screen, WHITE, resetButton)
        screen.blit(buttonText, buttonRect)
```

```
        # Exit button
        exitButton = pygame.Rect(
            (2 / 3) * width + BOARD_PADDING, (1 / 3) * height + 90,
            (width / 3) - BOARD_PADDING * 2, 50
        )
        buttonText = mediumFont.render("Exit", True, BLACK)
        buttonRect = buttonText.get_rect()
        buttonRect.center = exitButton.center
        pygame.draw.rect(screen, WHITE, exitButton)
        screen.blit(buttonText, buttonRect)

        # Display text
        text = "Lost" if lost else "Won" if len(marked) == HEIGHT * WIDTH - MINES else ""
        text = mediumFont.render(text, True, WHITE)
        textRect = text.get_rect()
        textRect.center = ((5 / 6) * width, (3 / 4) * height)
        screen.blit(text, textRect)

        move = None

        left, _, right = pygame.mouse.get_pressed()

        # Check for a right-click to toggle flagging
        if right == 1 and not lost:
            mouse = pygame.mouse.get_pos()
            for i in range(HEIGHT):
                for j in range(WIDTH):
                    if cells[i][j].collidepoint(mouse) and (i, j) not in marked:
                        if (i, j) in flags:
```

```
                    flags.remove((i, j))
                else:
                    flags.add((i, j))
                time.sleep(0.2)

    elif left == 1:
        mouse = pygame.mouse.get_pos()

        # If AI button clicked, make an AI move
        if aiButton.collidepoint(mouse) and not lost:
            move = ai.make_move(toPrint=False)
            if move is None:
                flags = ai.mines.copy()
                print("No moves left to make.")
            else:
                print("AI making safe move.")
            time.sleep(0.2)

        # Reset game state
        elif resetButton.collidepoint(mouse):
            game = GameControl(height=HEIGHT, width=WIDTH, mines=MINES)
            ai = Player(game, height=HEIGHT, width=WIDTH)
            marked = set()
            flags = set()
            lost = False
            continue

        elif exitButton.collidepoint(mouse):
            sys.exit()
```

```
        # User-made move
        elif not lost:
            for i in range(HEIGHT):
                for j in range(WIDTH):
                    if (cells[i][j].collidepoint(mouse)
                            and (i, j) not in flags
                            and (i, j) not in marked):
                        move = (i, j)

    # Make move and update AI knowledge
    if move:
        if move in ai.mines:
            flags.add(move)
        if game.is_mine(move):
            lost = True
        else:
            nearby = game.hint(move)
            marked.add(move)
            ai.matching()
            ai.insert_new_clause(move, nearby)
    else:
        ai.matching()
    # let the later win can show the flags on the game board
    if len(marked) + MINES == HEIGHT * WIDTH:
        flags = game.mines.copy()
        lost = False

    pygame.display.flip()
```