Lab3 report

# Implementation

Resource: Minesweeper - CS50's Introduction to Artificial Intelligence with Python (harvard.edu)

In this assignment, I reference the game structure base on the code Harvard university provided online. It has done the gui part of minesweeper and left the logic implementation part for us, so it's more convenient for us to debug and visualization can help us to better realize the result.

**# Game control**

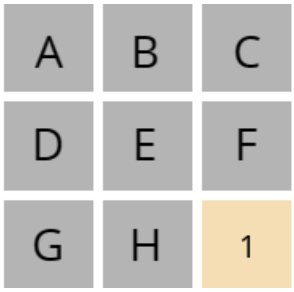
There are three difficulty levels for the user to choose, which are

- Easy
  - Gameboard size: 9*9
  - Mines: 10
- Medium
  - Gameboard size: 16*16
  - Mines: 25
- Hard
  - Gameboard size: 16*30
  - Mines: 99

The amount of initial safe cells is round(sqrt(Gameboard size))

**# Game logic**

After consulting with the teacher, he mentioned that it is allowed that we use method other than CNF to complete this assignment, so I use a different way to represent the clause in knowledge base, which is a less complicated expression than propositional logic.



Take this picture as example, we can use the knowledge from the lower left square and generate a sentence {E, F, H} = 1, then insert it into the knowledge base, rather than generate C(3,3) + C(3,2) clauses.

**# Game Flow**

Frist choose a cell that hasn't be chose before from a set named safe which contains all the safe cell, then use function add_knowledge to check operations such as subsumption and duplication after generating a new sentence with knowledge of this cell.

```
neighbors, count = self.get_cell_neighbors(cell, count)
sentence = Sentence(neighbors, count)
self.knowledge.append(sentence)
```

```
# n == 0
if count == 0:
    self.knowledge.pop(-1)
    sentence = None
    for n in neighbors:
        self.mark_safe(n)
# n == m
elif len(neighbors) == count:
    self.knowledge.pop(-1)
    sentence = None
    for n in neighbors:
        self.mark_mine(n)

# m > n > 0
if sentence is not None:
    new_inferences = []
    for s in self.knowledge:
        self.matching(s, sentence, new_inferences)

    self.knowledge.extend(new_inferences)
```

When n==0, we can mark all of cell's neighbor as safe cell.

When n==m, mark all of cell's neighbor as mine.

When m>n>0, use function matching to handle such situation.

```
def matching(self, s1, s2, new_inferences):
    if s1 != s2:
        if s1.cells.issuperset(s2.cells):
            sdiff = s1.cells-s2.cells
            # safes
            if s1.count == s2.count:
                for safe in sdiff:
                    self.mark_safe(safe)
            # mines
            elif len(sdiff) == s1.count - s2.count:
                for mine in sdiff:
                    self.mark_mine(mine)
            # subsumption
            else:
                new_inferences.append(
                    Sentence(sdiff, s1.count - s2.count)
                )
```

Function matching is used to complete subsumption when the element of a sentence

is another sentence's subset, when this happens, we can generate a stricter clause or know mines or safe cells position directly. For example, if s1 is {A, B, C, D} = 2 and s2 is {B, C, D} = 2, we can inference that A is a safe cell and mark it. Another example is when s1 is {A, B, C, D, E} = 3 and s2 is {C, D, E} =2, we can inference a new sentence s3, which is {A, B} = 1 and add it into our knowledge base.

```
new_inferences = []
for s1 in self.knowledge:
    for s2 in self.knowledge:
        self.matching(s1, s2, new_inferences)

self.knowledge.extend(new_inferences)
```

Doing matching again between every two sentences in knowledge base. Spec suggest us to do pairwise matching when there is no valid move, but I do it every time a move is make, so when there's no valid move to make, we don't have to wait and stuck for several iterations but can be sure that there are no moves that can be inference.

```
# handle duplication
ukb = []
for s in self.knowledge:
    if s not in ukb:
        ukb.append(s)
self.knowledge = ukb
```

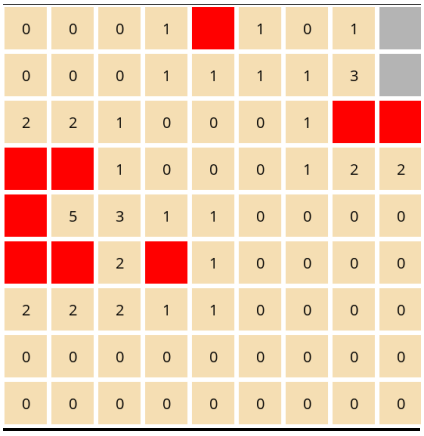
Deal with duplication at last.

# Discussion

**# Stuck**

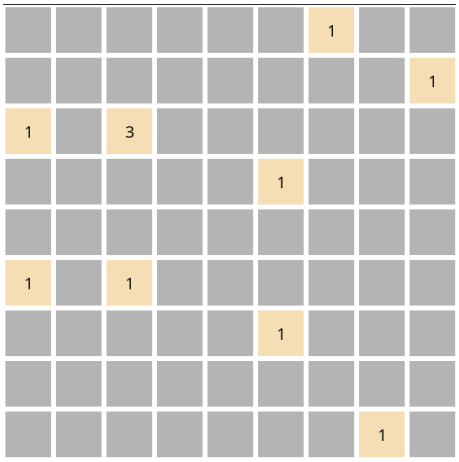
There are several situations which stuck may happen.

(gray means unmark, light yellow means marked safe cell, red means marked mine)

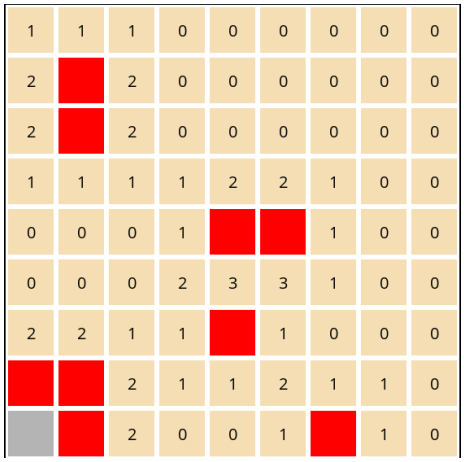


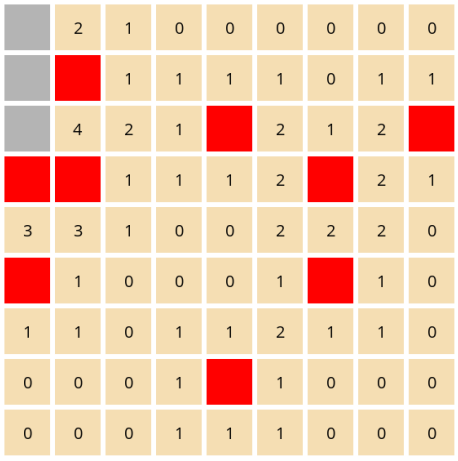

The first one is situation as above figure, from which we can see that there's only one

mine left and there's two cells left, so we cannot do any inference to get possible move but only guess.



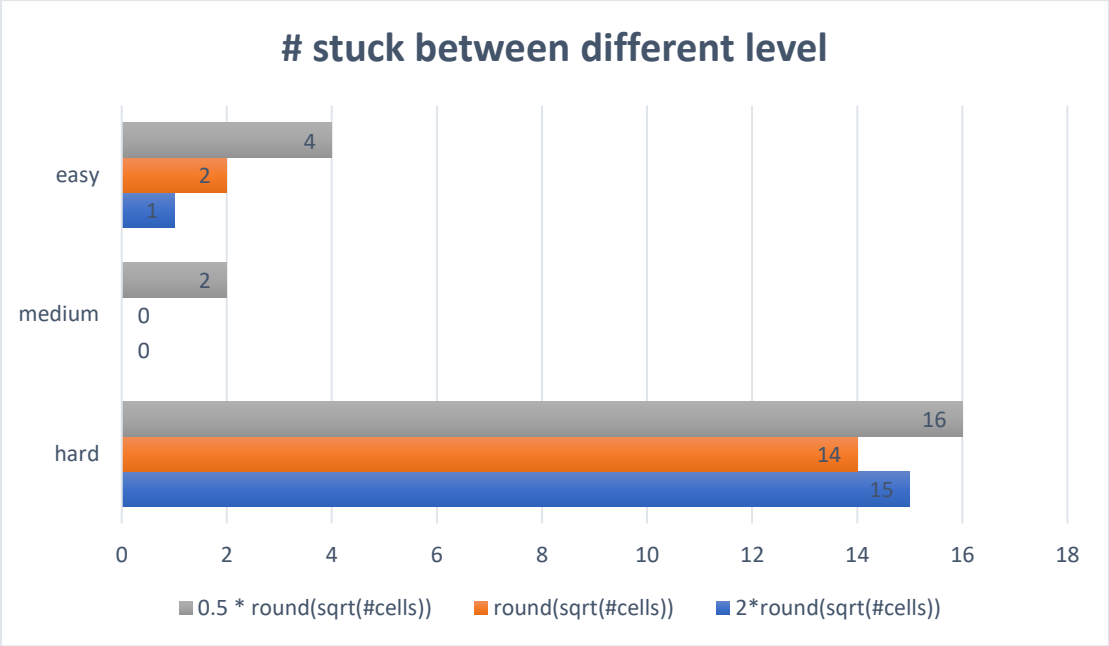
The second one happens when game just started, since the initial safe cells are randomly selected, there are chances that we are not able to make further prediction after the initial safe cell has all been marked because every safe cell is separated.





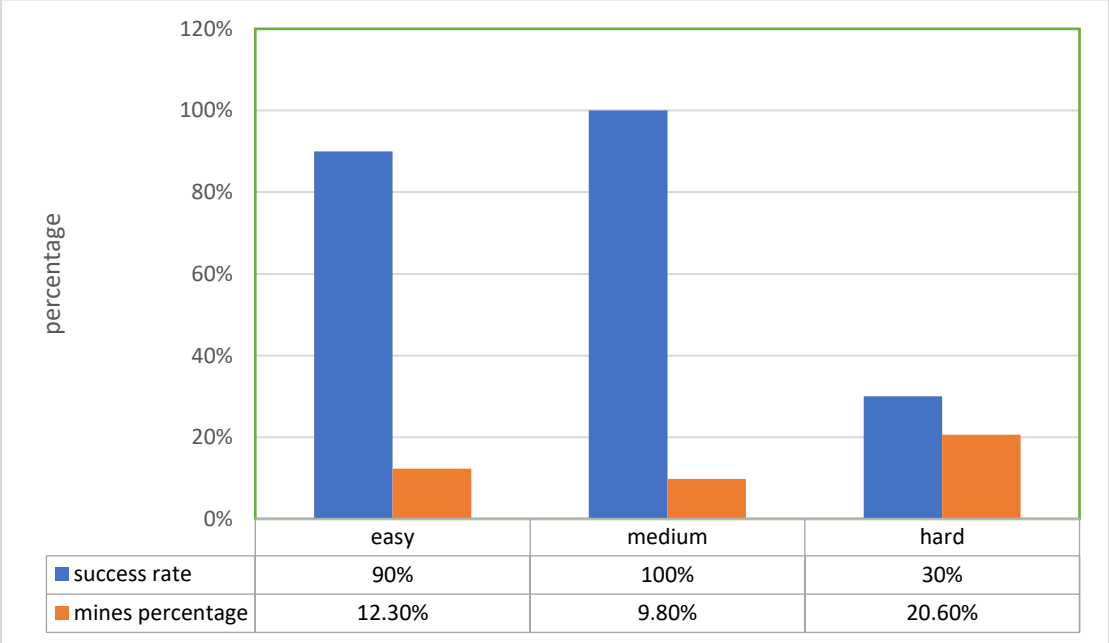
These situations are also possible because we don't implement endgame to check if the game has reached its end state. In the case of left figure, based on our logic agent, we don't not have any information about the last cell before, so there won't be a valid move for it to make. In the case of right figure, assume the cell from up to down is A,B,C, our knowledge base now has {A, B} = 1 and {B, C} = 1, but without the check of endgame, we can't know that {A, B, C} = 1 and thus inference A is safe and C is safe.

**# different initial safe cell**




In this experiment I test the number of stuck times out of 20 games. From the above figure we can find that as the number of initial safe cells increases, the number of stuck also decreases. But it's less obvious as the difficulty levels increases, it might because that the proportion of safe cells in the overall grid decreases as the difficulty increases.(0.111, 0.0625, 0.046)

**# success rate of different difficulty**




From the above figure we can also found that there is a negative correlation between

winning percentage and mines percentage out of the board. In other words, as the percentage of mines increases, the chances of winning the game tend to decrease. This might happen because as the mine's percentage increases, the number of potential mine locations also increases. This leads to a higher complexity in reasoning for logical agents. The agents encounter situations where multiple cells can potentially contain mines, and the available information becomes less conclusive. When the mines percentage out of the board is low, there is a higher likelihood of encountering safe cells or cells with fewer neighboring mines. This provides logical agents with more precise and reliable information to make informed decisions. They can use logical rules to inference sentence into the knowledge base more often.

# Conclusion

In this assignment, I use logical agents to solve minesweeper. In the beginning, it's difficult for me to understand what the spec want us to do, but after searching some information about propositional logic and how logical agent are implemented, I became more aware of what's the logic of the whole game flow and why would generate such clause a cell is chosen. I think it's a great opportunity for us to apply logical reasoning skill, leverage various logical rules and strategies, and witness the agent's ability to make informed decisions in minesweeper.

# Appendix

```python
import pygame
import sys
import time
import numpy as np
import random

from minesweeper import Minesweeper, MinesweeperAI

difficulty = input()
if difficulty == "easy":
    HEIGHT = 9
    WIDTH = 9
    MINES = 10
elif difficulty == "medium":
    HEIGHT = 16
    WIDTH = 16
    MINES = 25
elif difficulty == "hard":
    HEIGHT = 16
    WIDTH = 30
    MINES = 99

# Colors
BLACK = (0, 0, 0)
GRAY = (180, 180, 180)
WHITE = (255, 255, 255)
PINK = (255, 192, 203)
RED = (255, 0, 0)

# Create game
pygame.init()
size = width, height = 1500, 600
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)
```

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

safe_cell = []
print(np.round(np.sqrt(HEIGHT*WIDTH)))
while len(safe_cell) < 0.5 * np.round(np.sqrt(HEIGHT*WIDTH)):
    cell = (random.randint(0, HEIGHT-1), random.randint(0, WIDTH-1))
    if cell not in safe_cell and not game.is_mine(cell):
        cnt = game.nearby_mines(cell)
        ai.add_knowledge(cell, cnt)
        safe_cell.append(cell)
        revealed.add(cell)

# Show instructions initially
instructions = True

# Autoplay game
autoplay = False
autoplaySpeed = 0
makeAiMove = False

# Show Safe and Mine Cells
showInference = False

stuck = 0
reset = False
count = 0

while True:

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

    screen.fill(BLACK)
```

```
    if game.mines == flags or reset==True:
        count += 1
        print(count)
        if count >= 20:
            break
        reset = False
        game = Minesweeper(height=HEIGHT, width=WIDTH, mines=MINES)
        ai = MinesweeperAI(height=HEIGHT, width=WIDTH)
        revealed = set()
        flags = set()
        lost = False
        mine_detonated = None

        safe_cell = []
        while len(safe_cell) < np.round(np.sqrt(HEIGHT*WIDTH)):
            cell = (random.randint(0, HEIGHT-1), random.randint(0, WIDTH-1))
            if cell not in safe_cell and not game.is_mine(cell):
                cnt = game.nearby_mines(cell)
                ai.add_knowledge(cell, cnt)
                safe_cell.append(cell)
                revealed.add(cell)
```

```
        # Reset game state
        elif resetButton.collidepoint(mouse):
            game = Minesweeper(height=HEIGHT, width=WIDTH, mines=MINES)
            ai = MinesweeperAI(height=HEIGHT, width=WIDTH)
            revealed = set()
            flags = set()
            lost = False
            mine_detonated = None

            safe_cell = []
            while len(safe_cell) < np.round(np.sqrt(HEIGHT*WIDTH)):
                cell = (random.randint(0, HEIGHT-1), random.randint(0, WIDTH-1))
                if cell not in safe_cell and not game.is_mine(cell):
                    cnt = game.nearby_mines(cell)
                    ai.add_knowledge(cell, cnt)
                    safe_cell.append(cell)

            continue
```

```
if autoplay or makeAiMove:
    if makeAiMove:
        makeAiMove = False
    move = ai.make_safe_move()
    if move is None:
        move = ai.make_random_move()
        if move is None:
            flags = ai.mines.copy()
            print("No moves left to make.")
            # autoplay = False
        else:
            print("No known safe moves, AI stuck.")
            move = None
            # autoplay = False
            stuck += 1
            reset = True
    else:
        print("AI making safe move.")

    # Add delay for autoplay
    if autoplay:
        time.sleep(autoplaySpeed)

# Make move and update AI knowledge
if move:
    if game.is_mine(move):
        lost = True
        mine_detonated = move
        autoplay = False
    else:
        nearby = game.nearby_mines(move)
        revealed.add(move)
        ai.add_knowledge(move, nearby)

pygame.display.flip()
nt("stuck", stuck)
```

```python
def mark_mine(self, cell):
    """
    Marks a cell as a mine, and updates all knowledge
    to mark that cell as a mine as well.
    """
    self.mines.add(cell)
    for sentence in self.knowledge:
        sentence.mark_mine(cell)

def mark_safe(self, cell):
    """
    Marks a cell as safe, and updates all knowledge
    to mark that cell as safe as well.
    """
    self.safes.add(cell)
    for sentence in self.knowledge:
        sentence.mark_safe(cell)
```

```python
def add_knowledge(self, cell, count):
    """
    Called when the Minesweeper board tells us, for a given
    safe cell, how many neighboring cells have mines in them.

    This function should:
        1) mark the cell as a move that has been made
        2) mark the cell as safe
        3) add a new sentence to the AI's knowledge base
           based on the value of `cell` and `count`
        4) mark any additional cells as safe or as mines
           if it can be concluded based on the AI's knowledge base
        5) add any new sentences to the AI's knowledge base
           if they can be inferred from existing knowledge
    """
    # Mark cell as safe and add to moves_made
    self.mark_safe(cell)
    self.moves_made.add(cell)

    # Create and Add sentence to knowledge
    neighbors, count = self.get_cell_neighbors(cell, count)
    sentence = Sentence(neighbors, count)
    self.knowledge.append(sentence)

    # n == 0
    if count == 0:
        self.knowledge.pop(-1)
        sentence = None
        for n in neighbors:
            self.mark_safe(n)
    # n == m
    elif len(neighbors) == count:
        self.knowledge.pop(-1)
        sentence = None
        for n in neighbors:
            self.mark_mine(n)

    # m > n > 0
    if sentence is not None:
        new_inferences = []
        for s in self.knowledge:
            self.matching(s, sentence, new_inferences)

        self.knowledge.extend(new_inferences)

    new_inferences = []
    for s1 in self.knowledge:
        for s2 in self.knowledge:
            self.matching(s1, s2, new_inferences)

    self.knowledge.extend(new_inferences)
```

```
# m > n > 0
if sentence is not None:
    new_inferences = []
    for s in self.knowledge:
        self.matching(s, sentence, new_inferences)

    self.knowledge.extend(new_inferences)

new_inferences = []
for s1 in self.knowledge:
    for s2 in self.knowledge:
        self.matching(s1, s2, new_inferences)

self.knowledge.extend(new_inferences)

# handle duplication
ukb = []
for s in self.knowledge:
    if s not in ukb:
        ukb.append(s)
self.knowledge = ukb
```

```
def matching(self, s1, s2, new_inferences):
    if s1 != s2:
        if s1.cells.issuperset(s2.cells):
            sdiff = s1.cells-s2.cells
            # safes
            if s1.count == s2.count:
                for safe in sdiff:
                    self.mark_safe(safe)
            # mines
            elif len(sdiff) == s1.count - s2.count:
                for mine in sdiff:
                    self.mark_mine(mine)
            # subsumption
            else:
                new_inferences.append(
                    Sentence(sdiff, s1.count - s2.count)
                )
        elif s2.cells.issuperset(s1.cells):
            sdiff = s2.cells-s1.cells
            # safes
            if s2.count == s1.count:
                for safe in sdiff:
                    self.mark_safe(safe)
            # mines
            elif len(sdiff) == s2.count - s1.count:
                for mine in sdiff:
                    self.mark_mine(mine)
            # subsumption
            else:
                new_inferences.append(
                    Sentence(sdiff, s2.count - s1.count)
                    )
```

```
def make_safe_move(self):
    """
    Returns a safe cell to choose on the Minesweeper board.
    The move must be known to be safe, and not already a move
    that has been made.

    This function may use the knowledge in self.mines, self.safes
    and self.moves_made, but should not modify any of those values.
    """
    safeCells = self.safes - self.moves_made
    if safeCells:
        print('Making a Safe Move! Safe moves available: ', len(safeCells))
        return random.choice(list(safeCells))

    return None
```