Project 3 Report AI Capstone

Name: Kai-Jie Lin
Student ID: 110652019
May 8, 2023

1 Method

1.1 Game Environment

Initialize the board with a preset number of mines. Provid three different level of the Minesweeper are: Easy (9x9 board with 10 mines), Medium (16x16 board with 25 mines), and Hard (30x16 board with 99 mines). Provide the hint (number of surrounding mines) when queried for a cell by the player module. Provide $\sqrt{N}$ safe cells. N is the number of board cells.
Game termination:
The agent win when all mines are discovered, lose the game when move on a bomb cell. The game is stuck when no new markings can be made as there are multiple solutions to the unmarked cells.

1.2 Algorithm

Knowledge Representation

The AI's knowledge is represented as the following logical sentence: {A, B, C, D, E, F, G, H} = 1 where {A, B, C etc.} are a set of cells, and the number 1 is the count of mines among those cells. This representation allows the following inferences to be made, e.g.: {D, E} = 0. This implies that none of D, E contain mines, i.e. all are safe cells. {A, B, C} = 3 This implies that all cells A, B, C contain a mine. Furthermore, in general when we have two sentences where sentence A is a subset of sentence B, a new sentence B - A can be infered. Hence while playing minesweeper and clicking on cells, logical sentences are added to the AI's knowledge base. Often as a new sentence is added to the knowledge base, further inferences can be made allowing the identification of mines or safe spaces.

In addition to the basic representation of a set of cells and the number of mines within that set, there are other logical representations that can be used in minesweeper. One common technique is to represent each cell as a propositional variable that can take on one of two values: true or false, representing whether or not the cell contains a mine. This allows the AI to reason about the state of each cell independently, rather than treating them as a group.

Another important aspect of knowledge representation in minesweeper is the concept of constraints. For example, if the AI knows that a certain row contains two mines and there are only two unmarked cells in that row, then those cells must contain mines. These constraints can be used to infer the state of other cells in the game, even if there is no direct information available about those cells.

One challenge in representing knowledge in minesweeper is the potential for conflicting information. If the AI receives two logical sentences that are mutually exclusive, such as A = 1 and A = 0, it must resolve the conflict in order to make further inferences. This can be done using techniques such as constraint propagation or resolution, which allow the AI to identify inconsistencies and adjust its knowledge base accordingly.

Overall, the choice of knowledge representation in minesweeper will depend on the specific implementation and goals of the AI system. However, regardless of the representation used, the ability to reason about the state of cells and infer new information from logical sentences is critical for success in the game.

Sentence(Clasuse) Module

The Sentence class in the implementation of the game contains functions for manipulating logical sentences.

1. The known_mines function should return a set of all of the cells in self.cells that are known to be mines.

2. The known_safes function should return a set of all the cells in self.cells that are known to be safe.

3. The mark_mine function should first check to see if cell is one of the cells included in the sentence.

    - If cell is in the sentence, the function should update the sentence so that cell is no longer in the sentence, but still represents a logically correct sentence given that cell is known to be a mine.

    - If cell is not in the sentence, then no action is necessary.

4. The mark_safe function should first check to see if cell is one of the cells included in the sentence.

    - If cell is in the sentence, the function should update the sentence so that cell is no longer in the sentence, but still represents a logically correct sentence given that cell is known to be safe.

    - If cell is not in the sentence, then no action is necessary.

Minesweeper AI Module

In the MinesweeperAI class, complete the implementations of add_knowledge, make_action.

1. add_knowledge (Resolution) should accept a cell (represented as a tuple (i, j)) and its corresponding count, and update self.mines, self.safes, self.moves_made, and self.knowledge with any new information that the AI can infer, given that cell is known to be a safe cell with count mines neighboring it.

    - The function should mark the cell as one of the moves made in the game.
    - The function should mark the cell as a safe cell, updating any sentences that contain the cell as well.

- The function should add a new sentence to the AI's knowledge base, based on the value of cell and count, to indicate that count of the cell's neighbors are mines. Be sure to only include cells whose state is still undetermined in the sentence.

2. make_action return a move (i,j) that is known to be safe, if there is no any safe move, then return a random move.

2 Experiment

Difficulty	**Win Rate(100 Rounds)**
Easy	100
Medium	100
Hard	36

Based on the experiment results, the Minesweeper AI agent performs extremely well in the Easy and Medium difficulty levels, with a 100

One possible reason for this could be that the Hard level has a much larger board with a higher number of mines, making it more challenging for the AI agent to make accurate inferences based on its current knowledge base. As a result, it may struggle to identify the locations of all the mines and may end up selecting a bomb cell, causing the game to be lost.

3 Conclusion and Future Work

The Minesweeper AI developed using logical inference and resolution is able to win the game with a high success rate in easy and medium levels but has a lower success rate in the hard level. The algorithm uses logical representation of knowledge to make inferences and update the knowledge base during the game. The performance in the hard level can be improved by incorporating more advanced techniques like probability distribution and machine learning. To improve the performance of the Minesweeper AI, more advanced techniques can be used like reinforcement learning, deep learning, or Monte Carlo simulations. Reinforcement learning can be used to train the agent to choose the best move based on rewards and punishments. Deep learning can be used to identify patterns in the game and make decisions based on the learned patterns. Monte Carlo simulations can be used to generate and evaluate different scenarios to choose the best move. These techniques can be combined to develop a more sophisticated Minesweeper AI that can perform well in all levels.

Appendix

3.1 Full implementation

```
    import random
import math

class Minesweeper:
    def __init__(self, row, col, mine_num):
        self.board = [[0 for _ in range(col)] for _ in range(row)] #True if
            mine in (i,j), False if none
        self.cell_show = [[False for _ in range(col)] for _ in range(row)]
        self.row = row
        self.col = col
        self.mines = set()

        #add mines randomly
        while len(self.mines) != mine_num:
            i = random.randrange(row)
            j = random.randrange(col)
            if (i,j) not in self.mines:
                self.mines.add((i,j))
                self.board[i][j] = -1

        #mines found by player
        self.mines_found = set()

    def cell_val(self, cell):
        return self.board[cell[0]][cell[1]]

    def play_on(self, cell):
        self.board[cell[0]][cell[1]] = self.count_mine(cell)
        self.cell_show[cell[0]][cell[1]] = True

    def Give_init_cell(self):
        # To start, use round(sqrt(#cells)) as the number of initial safe cells
            .
        init_num = round(math.sqrt(self.row * self.col))
        init_cell = []
        while len(init_cell) != init_num:
            i = random.randrange(self.row)
            j = random.randrange(self.col)
            if self.board[i][j] == 0:
                self.play_on((i,j))
                init_cell.append((i,j))
        return init_cell

    def is_mine(self, cell):
        i, j = cell
        return self.board[i][j] == -1
```

```
    def Show_Board(self, debug = False):
        for i in range(self.row):
            print("--" * self.col + "-")
            for j in range(self.col):
                if debug:
                    print(f"|{self.board[i][j]}", end="")
                elif self.cell_show[i][j]:
                    print(f"|{self.board[i][j]}", end="")
                else:
                    print("|*", end="")
            print("|")
        print("--"*self.col + "-")

    def count_mine(self, cell):
        # Compute number of mines nearby
        count = 0
        for i in range(cell[0]-1, cell[0]+2):
            for j in range(cell[1]-1, cell[1]+2):
                if (i,j)==cell:
                    continue
                if 0 <= i < self.row and 0 <= j < self.col:
                    count += 1 if self.board[i][j]==-1 else 0
        return count

    def check_win(self):
        return self.mines_found == self.mines

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):
        self.cells = set(cells)
        self.count = count

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

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

    def known_mines(self):
        """
        Returns the set of all cells in self.cells known to be mines.
        """
```

```
        # If count of mines is equal to number of cells (and > 0), all cells
            are mines:
        if len(self.cells) == self.count and self.count != 0:
            #print('Mine Identified! - ', self.cells)
            return self.cells
        else:
            return set()

    def known_safes(self):
        """
        Returns the set of all cells in self.cells known to be safe.
        """

        # If count of mines is zero then all cells in the sentence are safe:
        if self.count == 0:
            return self.cells
        else:
            return set()

    def mark_mine(self, cell):
        """
        Updates internal knowledge representation given the fact that
        a cell is known to be a mine.
        """

        # If cell is in the sentence, remove it and decrement count by one
        if cell in self.cells:
            self.cells.remove(cell)
            self.count -= 1

    def mark_safe(self, cell):
        """
        Updates internal knowledge representation given the fact that
        a cell is known to be safe.
        """

        # If cell is in the sentence, remove it, but do not decrement count
        if cell in self.cells:
            self.cells.remove(cell)

class Player:
    def __init__(self, row, col, mine_num):
        self.row = row
        self.col = col
        self.tot_mine_num = mine_num

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

        # Keep track of cells known to be safe or mines
```

```
        self.mines = set()
        self.safes = set()

        # List of sentence about the game
        self.knowledge = []

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

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

    def add_knowledge(self, cell, count):
        # Mark the cell as a move that has been made, and mark as safe:
        self.action_made.add(cell)
        self.mark_safe(cell)

        # Create set to store undecided cells for KB:
        new_sentence_cells = set()

        # Loop over all cells within one row and column
        for i in range(cell[0] - 1, cell[0] + 2):
            for j in range(cell[1] - 1, cell[1] + 2):

                # Ignore the cell itself
                if (i, j) == cell:
                    continue

                # If cells are already safe, ignore them:
                if (i, j) in self.safes:
                    continue

                # If cells are known to be mines, reduce count by 1 and ignore
                    them:
                if (i, j) in self.mines:
                    count = count - 1
                    continue

                # Otherwise add them to sentence if they are in the game board:
                if 0 <= i < self.row and 0 <= j < self.col:
                    new_sentence_cells.add((i, j))

        # Add the new sentence to the AI's Knowledge Base:
        #print(f'Move on cell: {cell} has added sentence to knowledge {
            new_sentence_cells} = {count}' )
        self.knowledge.append(Sentence(new_sentence_cells, count))
```

```
# Iteratively mark guaranteed mines and safes, and infer new knowledge:
knowledge_changed = True

while knowledge_changed:
    knowledge_changed = False

    safes = set()
    mines = set()

    # Get set of safe spaces and mines from KB
    for sentence in self.knowledge:
        safes = safes.union(sentence.known_safes())
        mines = mines.union(sentence.known_mines())

    # Mark any safe spaces or mines:
    if safes:
        knowledge_changed = True
        for safe in safes:
            self.mark_safe(safe)
    if mines:
        knowledge_changed = True
        for mine in mines:
            self.mark_mine(mine)

    # Remove any empty sentences from knowledge base:
    empty = Sentence(set(), 0)

    self.knowledge[:] = [x for x in self.knowledge if x != empty]

    # Try to infer new sentences from the current ones:
    for sentence_1 in self.knowledge:
        for sentence_2 in self.knowledge:

            # Ignore when sentences are identical
            if sentence_1.cells == sentence_2.cells:
                continue

            if sentence_1.cells == set() and sentence_1.count != 0:
                print('Error - sentence with no cells and count created
                    ')
                raise ValueError

            # Create a new sentence if 1 is subset of 2, and not in KB:
            if sentence_1.cells.issubset(sentence_2.cells):
                new_sentence_cells = sentence_2.cells - sentence_1.
                    cells
                new_sentence_count = sentence_2.count - sentence_1.
                    count
```

```
                    new_sentence = Sentence(new_sentence_cells,
                        new_sentence_count)

                    # Add to knowledge if not already in KB:
                    if new_sentence not in self.knowledge:
                        knowledge_changed = True
                        #print('New Inferred Knowledge: ', new_sentence, '
                            from', sentence_1, ' and ', sentence_2)
                        self.knowledge.append(new_sentence)

        # Print out AI current knowledge to terminal:
        #print('Current AI KB length: ', len(self.knowledge))
        #print('Known Mines: ', self.mines)
        #print('Safe Moves Remaining: ', self.safes - self.action_made)
        #print('==================================================')

    def make_action(self):
        safe_actions = self.safes - self.action_made
        if safe_actions:
            #print('Making a Safe Move! Safe moves available: ', len(
                safe_actions))
            return random.choice(list(safe_actions))

        # if no guaranteed safe moves can be made, then make random move
        # dictionary to hold possible moves and their mine probability:
        actions = {}
        MINES = self.tot_mine_num

        # Calculate basic probability of any cell being a mine with no KB:
        num_mines_left = MINES - len(self.mines)
        spaces_left = (self.row * self.col) - (len(self.action_made) + len(self
            .mines))

        # If no spaces are left that are acceptable moves, return no move
            possible
        if spaces_left == 0:
            return None

        basic_prob = num_mines_left / spaces_left

        ## Get list of all possible moves that are not mines
        for i in range(0, self.row):
            for j in range(0, self.col):
                if (i, j) not in self.action_made and (i, j) not in self.mines:
                    actions[(i, j)] = basic_prob

        # If no moves have been made (nothing in KB) then any is a good option:
        if actions and not self.knowledge:
            move = random.choice(list(actions.keys()))
            #print('AI Selecting Random Move With Basic Probability: ', move)
```

```
            return move

        # Otherwise can potentially improve random choice using KB:
        elif actions:
            for sentence in self.knowledge:
                num_cells = len(sentence.cells)
                count = sentence.count
                mine_prob = count / num_cells
                # If mine probabilty of each cell is worse than listed, update
                    it:
                for cell in sentence.cells:
                    if actions[cell] < mine_prob:
                        actions[cell] = mine_prob

            # Get and return random move with lowest mine probability:
            move_list = [[x, actions[x]] for x in actions]
            move_list.sort(key=lambda x: x[1])
            best_prob = move_list[0][1]

            best_moves = [x for x in move_list if x[1] == best_prob]
            move = random.choice(best_moves)[0]
            #print('AI Selecting Random Move with lowest mine probability using
                KB: ', move)

            # Return a random choice from the best moves list
            return move

def Run_Game(row, col, mine_num):
    Game = Minesweeper(row, col, mine_num)
    Ai = Player(row, col, mine_num)

    Init_safe_cells = Game.Give_init_cell()
    #print("Move on initial safe cell given by game board:")
    for cell in Init_safe_cells:
        Ai.add_knowledge(cell, Game.cell_val(cell))

    Game.Show_Board()
    end_state = 0
    episodes = 0
    print("===Game Start===")
    while not Game.check_win():
        action = Ai.make_action()
        episodes += 1
        if action == None:
            end_state = 1
            break
        if Game.cell_val(action) == -1:
            end_state = 2
            break
        Game.play_on(action)
```

```
            #Game.Show_Board()
            Ai.add_knowledge(action, Game.cell_val(action))
            Game.mines_found = Ai.mines


        Game.Show_Board(True)
        print(f"episodes: {episodes}")
        win = 0
        if end_state == 0:
            print("You WIN")
            win = 1
        elif end_state == 1:
            print("No move can be applied. Game Stuck!")
        else:
            print("You LOSE!")
        print("Final Board:")
        Game.Show_Board()
        return win


if __name__ == '__main__':
    Easy = (9,9,10)
    Medium = (16,16,25)
    Hard = (16, 30, 99)
    count = 0
    for i in range(100):
        count += Run_Game(*Hard)
    print(count)
```