COMPUTER SCIENCE DEPARTMENT - RUTGERS UNIVERSITY

CS 520 - Assignment 2

Bo Xu (bx49), Jiyu Zhang (jz644), Shijie Geng (sg1309)

October 29, 2017

1 QUESTIONS AND WRITE-UP

1.1 QUESTION 1: REPRESENTATION

Q: How did you represent the board in your program, and how did you represent the information / knowledge that clue cells reveal?

A: We basically use matrices to represent the board. There is an invisible (to the AI player) matrix that contains all the information of the board (positions of mines and number clues). Another visible matrix is for the AI player, which shows no information at the beginning stage. The AI will obtain information from the board by retrieving the visible matrix.
We use numbers to represent the clues revealed in the board, and store them in the board visible to the AI player.

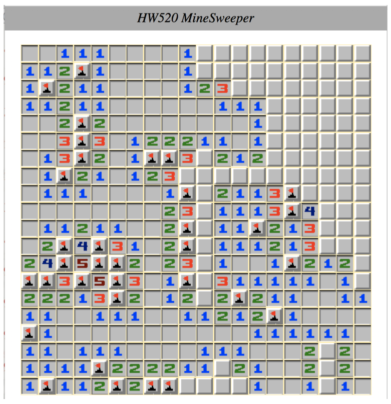

Figure 1.1: Our representation of minesweeper board

1.2 Question 2: Inference

Q: When you collect a new clue, how do you model / process / compute the information you gain from it? i.e., how do you update your current state of knowledge based on that clue? Does your program deduce everything it can from a given clue before continuing? If so, how can you be sure of this, and if not, how could you consider improving it?

A: Our AI computes the information that we have obtained mainly by two methods: *local search* and *tank search*. These two methods always find safe squares which must contain a mine or not.

Local search searches the board by fixed number of squares(9 in our implementation). There are two cases we can process safely:

Case 1: If the difference between the clue value and the number of flags equals to the number of unknown squares, then all the unknown squares must contain mines. For these squares, we put flags on them.

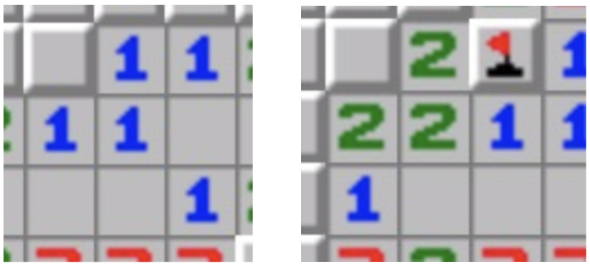

Figure 1.2: Case 1 examples - the left top square of each example must contain a mine

Case 2: If the difference between the clue value and the number of flags equals to 0, then all the remaining unknown squares must be safe and contain no mines. For these squares, we will uncover them.

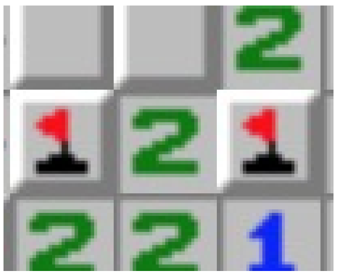

Figure 1.3: Case 1 examples - the left top square of each example must contain a mine

The Python script of local search is as follows:

Listing 1: Python Script for Local Search

```python
def local_search(self):
    if self.game.lose:
        return
    for i in range(self.height):
        for j in range(self.width):
            if self.game.lose:
                break
            # uk = unknown
            if self.visible_board[i][j] < 98:
                uk = 9
                clue_num = self.visible_board[i][j]
                e = [-1,0,1]
                uk_list = []
                for m in range(3):
                    for n in range(3):
                        if self.game.lose:
                            break
                        if i+e[m] < 0 or i+e[m] >= self.height or j+e[n] < 0 or j+e[n] >=
                            uk -= 1
                            continue
                        # there is a flag
                        if self.visible_board[i+e[m]][j+e[n]]==98:
                            clue_num -= 1
                            uk -= 1
                        # there is no mine
                        elif self.visible_board[i+e[m]][j+e[n]]<98:
                            uk -= 1
                        else:
                            uk_list.append([i+e[m], j+e[n]])
                self.clue_matrix[i][j] = clue_num
                if uk > 0:
                    # click all the uk
                    if clue_num == 0:
                        for x in range(len(uk_list)):
                            if self.game.lose:
                                break
                            y_index = uk_list[x][0]
                            x_index = uk_list[x][1]
                            if self.visible_board[y_index][x_index] == 99:
                                self.game.click(uk_list[x])
                                self.remain_squares -= 1
                                self.game.print_visible_board()
                                self.modification = True
                    # label all the uk
                    if uk == clue_num:
                        for x in range(len(uk_list)):
                            if self.game.lose:
                                break
                            y_index = uk_list[x][0]
                            x_index = uk_list[x][1]
                            if self.visible_board[y_index][x_index] == 99:
                                self.game.flag(uk_list[x])
                                self.remain_mines -= 1
                                self.game.print_visible_board()
                                self.modification = True
                    if uk > clue_num and clue_num > 0:
                        for x in range(len(uk_list)):
                            y_index = uk_list[x][0]
                            x_index = uk_list[x][1]
                            self.pmatrix[y_index][x_index] += clue_num/uk
                        self.uk_matrix[i][j].extend(uk_list)
```

Tank search, however, is able to find safe squares using more than one clue numbers.

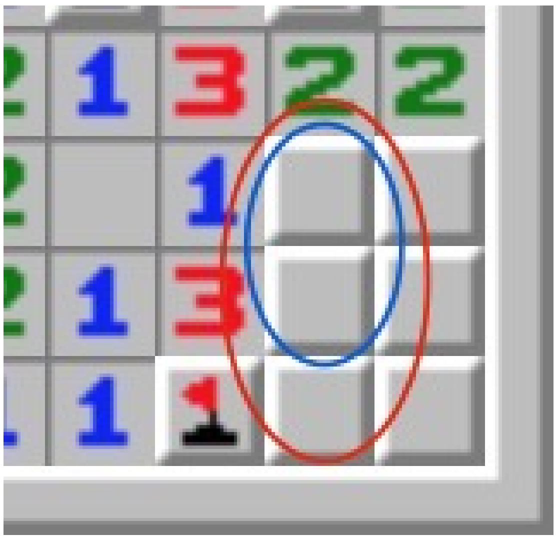

Figure 1.4: The red circle (encircle 3 squares) contains two mines. The blue circle (encircle 2 squares) which is a subset of the red one and contains a mine

From the above figure, we can conclude that the complement of the blue circle in the red circle must contain a mine.
The Python script of tank search is as follows:

Listing 2: Python Script for Tank Search

```
def tank_search(self):
    if self.game.lose or self.game.win:
        return
    for i in range(self.height):
        if self.game.lose or self.game.win:
            break
        for j in range(self.width):
            if self.game.lose or self.game.win:
                break
            if self.visible_board[i][j] < 98:
                e = [-1,0,1]
                for m in range(3):
                    for n in range(3):
                        if self.game.lose:
                            break
                        if i+e[m] < 0 or i+e[m] >= self.height or j+e[n] < 0 or j+e[n] >=
                            continue
                        lst1 = self.uk_matrix[i][j]
                        lst2 = self.uk_matrix[i+e[m]][j+e[n]]
                        if  sublist(lst1, lst2):
                            lst = [idx for idx in lst2 if idx not in lst1]
                            if self.clue_matrix[i+e[m]][j+e[n]] == 1 and self.clue_matrix
                                if len(lst)>0:
                                    for x in range(len(lst)):
                                        if self.game.lose:
                                            break
                                        y_index = lst[x][0]
                                        x_index = lst[x][1]
                                        if self.visible_board[y_index][x_index] == 99:
                                            self.game.click(lst[x])
                                            self.remain_squares -= 1
                                            self.game.print_visible_board()
                                            self.matrix_modification = True
                            if (self.clue_matrix[i+e[m]][j+e[n]] - self.clue_matrix[i][j]
                                if len(lst)>0:
                                    for x in range(len(lst)):
                                        if self.game.lose:
                                            break
                                        y_index = lst[x][0]
                                        x_index = lst[x][1]
                                        if self.visible_board[y_index][x_index] == 99:
                                            self.game.flag(lst[x])
                                            self.remain_mines -= 1
                                            self.game.print_visible_board()
                                            self.matrix_modification = True
```

Since we use local search to implement our AI player, our program only search part of the whole board. We don't go through all the possible configuration (which is very time-consuming). So our program doesn't deduce everything it can from a given clue. To improve this, we can directly use brute-force search. In this way, we can have a perfect deduction.

1.3 Question 3: Decisions

Q: Given a current state of the board, and a state of knowledge about the board, how does your program decide which cell to search next? Are there any risks, and how do you face them?

A: Based on the two search schemes above, our AI first chooses to uncover or mark all the safe squares which must contain a mine or must not. If there is no safe square remained, we can randomly choose a square to uncover from the remaining squares. Yes, this is risky. So we can use possibility to relieve the situation. For example, we can calculate the possibilities of all the surrounding squares, and choose to click the square with the lowest possibility to contain a mine.

1.4 QUESTION 4: PERFORMANCE I

Q: For a reasonably-sized board and a reasonable number of mines, include a play-by-play progression to completion or loss. Are there any points where your program makes a decision that you don't agree with? Are there any points where your program made a decision that surprised you? Why was your program able to make that decision?

A: In my opinion, the answer to both of the two questions is no. Because our search algorithms are deterministic, it doesn't improve itself based on its experiences. So the choice must be reasonable based on our algorithms.

1.5 QUESTION 5: PERFORMANCE II

Q: For a fixed, reasonable size of board, what is the largest number of mines that your program can still usually solve? Where does your program struggle?

A: For a 20×20 board, the largest number of mines that our program can still usually solve is 50. When our local search and tank search do not work well, our program will get in trouble since it only has the choice to click randomly.

1.6 QUESTION 6: EFFICIENCY

Q: What are some of the space or time constraints you run into in implementing this program? Are these problem specific constraints, or implementation specific constraints? In the case of implementation constraints, what could you improve on?

A: We don't encounter such a space or time constraints. Because our program utilizes the scheme of local search, so the space complexity is constant. As for the time complexity, it is based on the size of the board. After all, you need to explore (click or mark) all the squares in the board. And we don't have implementation constraints.

1.7 QUESTION 7: IMPROVEMENTS

Q: Consider augmenting your program's knowledge in the following way - when the user inputs the size of the board, they also input the total number of mines on the board. How can this information be modeled and included in your program, and used to inform action? How can you use this information to effectively improve the performance of your program, particularly in terms of the number of mines it can effectively solve.

A: We have already included the functions of inputting board size and mines number in our implementation. Since we don't use informed-search algorithms here, so in fact we don't use this information in our scheme.

1.8 AI PLAYING EXAMPLE

The example playing process of our AI player is showed as follows:

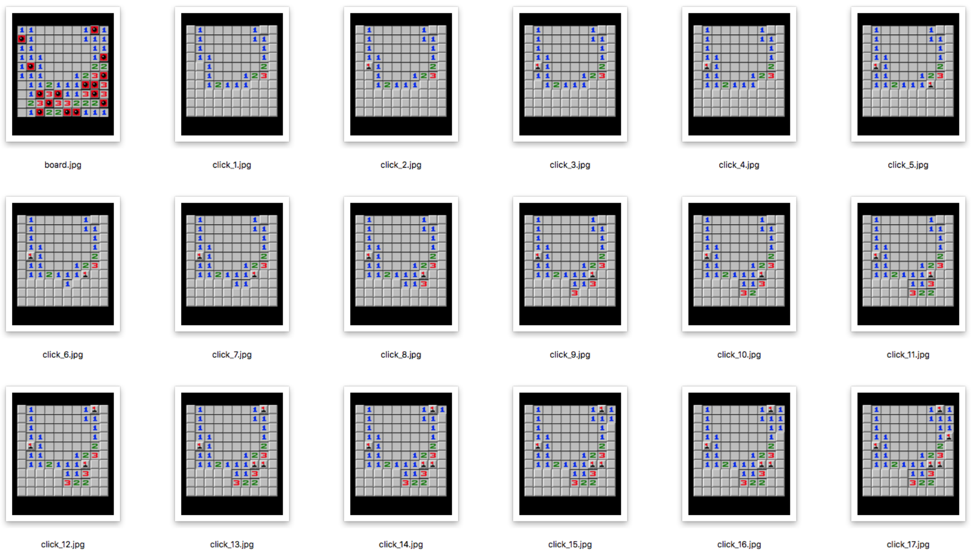

Figure 1.5: The example playing process of our AI player (Part I)

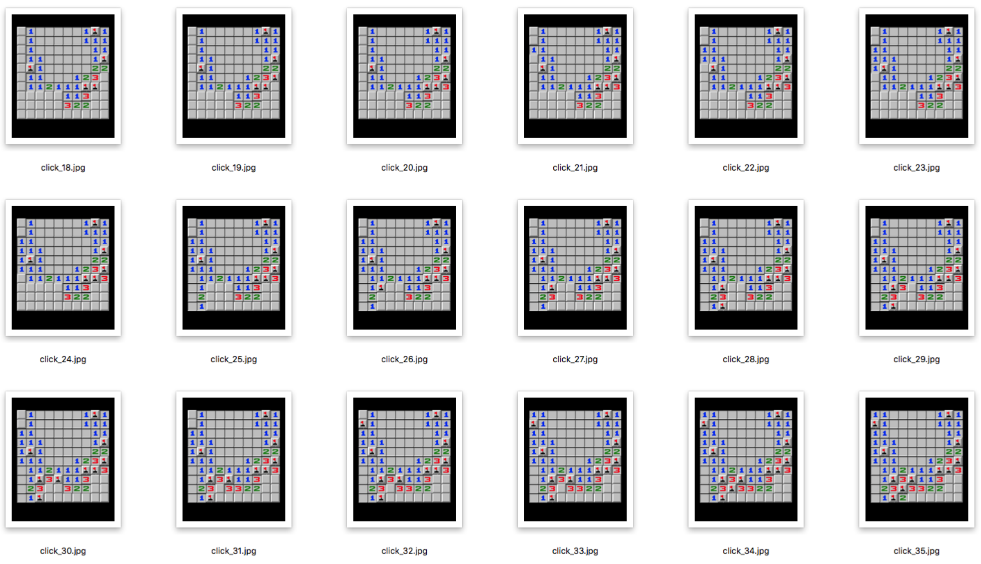

Figure 1.6: The example playing process of our AI player (Part II)

2 Bonus: Chains of Influence

2.1 **Question 1:**

Q: Based on your model and implementation, how can you characterize and build this chain of influence? Hint: What are some 'intermediate' facts along the chain of influence?
A: The chain of influence is in fact the same reasoning process as our local search and tank search scheme. So if the two schemes are able to keep working, the size of the chain will increase until we cannot find any safe squares. At this time, the chain of influence breaks and we have to choose the next square to click randomly.

2.2 **Question 2:**

Q: What influences or controls the length of the longest chain of influence when solving a certain board?
A: The size of the board, the number of mined squares, and the layout of all the mines are the three factors that controls the length of the longest chain of influence. If the layout of the mines is very tricky, the chain of influence can easily break and the AI player will need to collect information from other parts of revealed squares.

2.3 **Question 3:**

Q: How does the length of the chain of influence influence the efficiency of your solver?
A: The longer the chain of influence is, the more information we need to integrate to solve a special square. Therefore, long chains of influence will make our solver inefficient.

2.4 **Question 4:**

Q: Experiment. Can you find a board that yields particularly long chains of influence? How does this vary with the total number of mines?
A: Yes. See pictures below. In the first picture, to find the right top square labeled 1, our solver started at the bottom left part, then search along the fringe of squares already uncovered. Therefore, it went through all the squares in the board. This is the longest and the only chain of inference in the board. In the second picture, the solver began at the bottom right part. After it explored all the squares on the right of the red line, it chose to uncover one square in the left of the red line. This step, indicates that it cannot find any safe squares in the right part and choose square randomly, which means the chain of influence broke.
Therefore, the more mines are there in the board, the longer the chain of influence is.

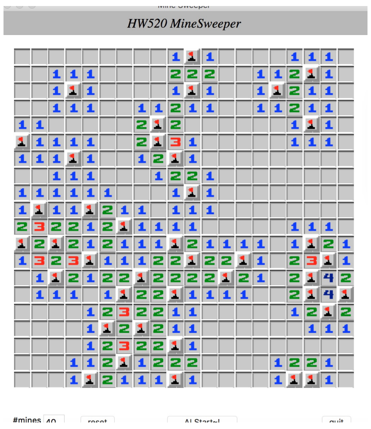

Figure 2.1: Example of long chains of influence I - 40 mines in the board

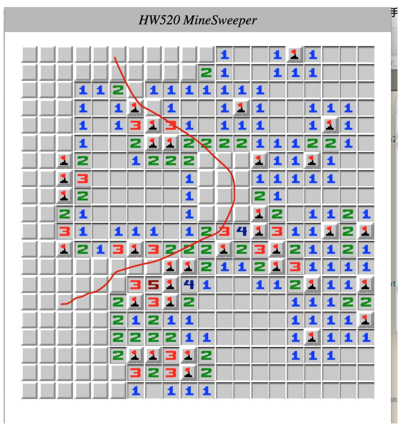

Figure 2.2: Example of long chains of influence II - 55 mines in the board

2.5 Question 5:

Q: Experiment. Spatially, how far can the influence of a given cell travel?
A: From the first picture above, we conclude that the longest chain of influence of the right top square will travel through all the squares in the board.

2.6 Question 6:

Q: Can you use this notion of minimizing the length of chains of influence to inform the decisions you make, to try to solve the board more efficiently?
A: From this point of view, we will use local search as often as possible. Because local search has a chain of influence of length 1. So we try to solve the squares that are the nearest from the uncovered part.

2.7 **Question 7:**

Q: Is solving minesweeper hard?
A: Yes, it is hard. Some papers have already proved that it is an NP-complete problem. To solve such problems, we don't have a polynomial time algorithm to efficiently solve it. And brute-force search with exponential time is the worse case.

3 Bonus: Dealing with Uncertainty

Q: How could you adapt your solver to each of these three cases? Try to implement at least one, and experiment with what kind of performance hit results, in terms of ability to clear the board.
A:We choose to implement the case when the clues underestimate the surrounding mines. For the cases that underestimate the number of surrounding mines. It chose squares boldly and was more likely to uncover a square with a mine, which decrease the change of success. Experimental data are showed in the table below.

Table 3.1: The successful rate of our AI player with or without handling uncertainty in 10×10 board - test for 100 times

Citation	5 mines	10 mines	15 mines	20 mines	25 mines
Normal	0.98	0.84	0.58	0.19	0.03
Underestimate	0.80	0.69	0.35	0.08	0.00