**CS 520: Assignment 2 – Minesweeper**

Group members:

|  |  |  |
| --- | --- | --- |
| Name | netID | Work |
| Yan GU | yg369 | Test and debug/Write-up of part2 & 3 |
| Yunfan LI | yl1269 | Visualization/Organization of program |
| Xueyu WU | xw318 | Minesweeper ALG/Organization of program/Write-up of part5 |
| Junlin LU | jl2364 | Minesweeper ALG/Write-up of part4 & 5 |

**2**

**Program Specification:**

Simulation of 10\*10 Grid with 10 mines

1. Run the app.py

图片包含 屏幕截图

已生成极高可信度的说明

1. Enter the rows and columns and number of mines and click *New Game*

图片包含 屏幕截图

已生成极高可信度的说明

1. Just click *Enter* for next step.

It is the first and second step.

图片包含 屏幕截图, 电子产品

已生成极高可信度的说明

Keep click *Enter*.

图片包含 屏幕截图, 顶部, 就坐, 监视器

已生成高可信度的说明

1. Keep click Enter, until a message box occur, then the game ends.

If all the mines are find, we win.

图片包含 屏幕截图, 计算机, 室内, 笔记本电脑

已生成极高可信度的说明

1. If we step into a mine or cannot traverse the whole board, we lose the game.

图片包含 屏幕截图

已生成极高可信度的说明

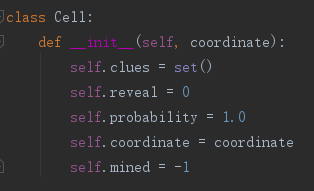
**3**

**Questions:**

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

We use a two-dimension numpy array (matrix) to represent the board, where cells with mine are value 1 and cells without mine are value 0.

We represent constraint of each cell with a class Cell with the following data structure:



In class Cell, *clues* are a set denotes the coordinate of unrevealed neighbors of a cell. *Reveal* is an int value which denotes the number of mines surrounding a cell, it is input by user. *Probability* denotes the probability of mine occurrence in a cell, which is initially set to 1 and modified by its parent cells. For example, if cell [0,0]’s reveal is 1, we can know cells [0,1] [1.0] [1,1] have a probability of 1/3. *Coordinate* is the coordinate of its own. *Mined* denotes whether the cell is a mine, if it is, then *mined* = 1, if not, then *mined* = 0, and initially we set it -1 for unknown cells which requires further search.

We build object of class *Cell* for each cell and use a matrix to store those objects. Each object is the constraint of a cell and the matrix *cells* contains all constraints in the whole board.

We represent the knowledge base in the following forms:

* Unknown\_cellt: set of all the unknown cells
* Clear\_cell: set of all the clear cells without mine
* Mine\_cell: set of all the cells with mine
* Clue\_cell: set of all the cells with a clue that is still useful, which means, if all the neighbors of a cell is clearly known, we remove the cell from clue\_cell set.
* Chains: would be explained in part 4.
* List cells: list of constraints.

1. **Inference: 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?**

* **STEP 0:** Basically, we set a rule, that is, we initially start the first step from the [0,0] and we set that the [0,0] does not contain a mine, so we will not fail in the first step. Because we think fail at the first step is not caused by our program, there is a probability that we randomly pick a cell at the first step, but we stand on a mine. So, in order to avoid that bad influence, we just use a easy way, pick [0,0] as first step and set [0,0] clear. The reason we choose a corner is below:

Let us consider the statistics. If the mine density is d, the probability of a particular square being clear is:

Prob{clear} = 1 – d

For squares that are not in a corner or on an edge of the board, the probability that that square is not a mine and does not have any adjacent mines is:

Prob{clear} = (1 – d)9

Similarly, for an edge square that is not in a corner:

Prob{clear} = (1 – d)6

and for a corner square:

Prob{clear} = (1 – d)4

Then we use the following strategies to maintain the knowledge base:

* **Step 1:** If the number of a cell (*reveal*) equals the number of unknowns near the cell, we can know all the nearby unknowns are mines, then we set *cell.reveal* = 0, and remove all the cells in the *cell.clues*. If the number of a cell is 0, all the nearby unknows are clear, and if number of a cell is less than the number of nearby unknowns, we use other strategies to narrow the scope of our clues.
* **STEP 2:** Given a set of non−trivial constraints, further simplification may be possible by noting that one constraint’s variables may be a subset of another constraint’s variables. For example, given a+b+c+d = 2 and b+c = 1, the former can be simplified to a+d = 1 and the latter left as b+c = 1. As a result of this simplification, some constraints may become trivial. If this happens, we will return to step 1 above. Note that during a typical game of minesweeper, the majority of the plays made in the game are a result of trivial constraints that are found in steps 1 and 2.
* **STEP 3:** Once all the solutions to all of the subsets of constraints have been found, the solutions can be analyzed to see if there are any cases where a square is known either to be a mine or to be clear (the variable is either 1 or 0 in all solutions, respectively). If such an instance is found, mines can be marked and/or squares probed with certainty of success. Note that marking mines that the solution set indicates are there with certainty does not provide any new information and therefore a guess may still be required in step 4 or later; however, if there are any clear squares implied by the solution set, they can be probed, the new constraints can be added to the constraint set, and the algorithm can immediately return to step 1 for a new round of simplifying the, now expanded, set of constraints.
* **STEP 4:** In step 3 it may be discovered that a coupled set of constraints requires a guess to be made and that there is no possibility of new information ever making the guess easier or eliminating it entirely. So we introduce a probability, *cell.probability*, we use it denotes the probability of a cell have a mine, and we use a queue that randomly pick the cell with least probability, I will discuss it further in the Question 3.

We have proved all the strategies above use the inference methods (∧, ∨, etc.), but it takes too long to put the proof here, so we choose not to write them down.

1. **Decisions: 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?**

When we go to a cell, we may discover a reveal number that tells us about the number of mines near this cell, and we can add the neighbors of this cell into a queue. This queue is the chain storing all the unknowns we are ready to search (in other ways, all the unknown neighbors of all the clear cells). And by using the inference in the Question 2, we can simplify all the clues such that all the clues will contain a reveal number and a set storing remaining-unknown neighbors of the cell. After dividing the reveal number with the number of remaining nearby unknowns, we assign a probability to each neighbor and add those neighbors into the priority queue, which arranged by comparing the reveal numbers we have collected, if some of cells have the same reveal numbers, we then compare the unknown neighbors of the cell and arrange the cell with least unknown neighbors at the top of the queue.

For example, when a cell is uncovered and the number reveals is not “good” enough to make a decision, such as a “1” appears in the upper left corner, at first, we just choose a cell from its neighbor since it is completely random. However, we soon found out that even the total number of mines is unknown, the number of mines in partial area is known, there is a probability comparison that we can employ to choose a “safer” step by applying priority queue. [1, 1]’s neighbors only have 1/5 of chances to be mine while [0, 0]’s neighbors have 50% to be a mine. Then the program tends to choose one of [1, 1]’s neighbors as the next step (Figure 3-1).

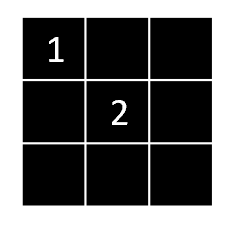


Figure 3-1

But stepping into diagonal cells with a lot of neighbors will grow the chain (This will demonstrate in part 4) and to be fair, there are too many cells that it doesn’t necessarily help us to determine the mines and the actual safe step. Therefore, in those cells with lowest mine probability, we decided to check the one with less uncover neighbors. As is shown in figure 3-2, [1, 0] only has 4 uncover neighbors while [1, 1] has 7, we should choose [1, 0], then we will get a smaller set that can easily be simplify. This is the basic strategy of dealing with uncertainty when solving minesweeper, that is trying to choose the cell with less potential uncertainty and more potential solvable logic. This way doesn’t reduce the probability of losing, as we can see each surrounding cell all have 1/3 of chances is mine, but it increases the probability of winning because our next step is more likely to get us a more solvable set.

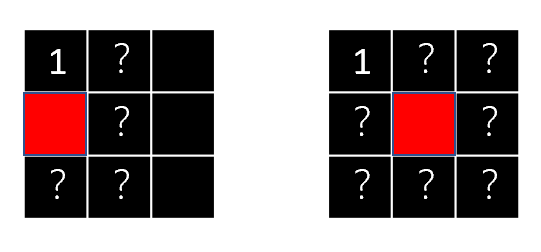


Figure 3-2

1. **Performance: 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?**

There is no situation that the program makes a decision which we don’t agree with or feel surprised. Because the strategies we made is exactly according to the strategies we use when we solve the minesweeper games by ourselves and we ensure that it will be the same strategies. The steps computer uses is what we use.

Which would be different is that when dealing with uncertainties, when human and computer randomly choose a cell, it may be different because it is ***randomly*** different.

1. **Performance: 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?**

For a fixed, reasonable size of board (we choose 10\*10 and 20\*20), when mines are roughly less than 15% of the grids, we will win the game for the most part, when mines are roughly 20%-25% of the grids, we may sometimes lose the game, when mines are more than 25% of the grids, we will usually lose the game.

Our program struggles when it won’t encounter a zero or a max reveal number, then we will use the uncertainty to solve the problem, at the point, when ratio of mines getting bigger, the probability for stepping into a mine getting bigger too. But we have to do that insecure move.

1. **Efficiency: 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?**

Both the space and time complexity are related with the maximum length of chain.

So, space complexity is **O(n2)**.

We take the worst situation as example, if n2 cells in the chain, for each time we get a clue, we update the former clues by comparison, so for n2 cells, the number of comparison is

82 \* ((12)2+(22)2+(32)2+……+(n2)2)

And we have at most n2 cells in chain, so we can approximately know that it may be O(n8). But it is not easy to get exactly real number k > 4 for O(nk).

So, we just say time complexity is **polynomial**.

And we can easily know both are **implementation specific constraints**.

**The improvement:** we already simplify the space for unknown and clue set to minimalist, but for time complexity, in order to get the simplest knowledge base, we must spend a lot time comparing clue sets. But we can still reduce the iteration by study about the relationship between two clue sets, such as exclusive and independent that can prevent some unnecessary comparison which will not simplify two clue sets.

1. **Improvements: 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?**

**The improved strategy:** we can change the strategy mentioned in Question 4 by calculating the probabilities of all the cells in the chain as well as the probabilities of the cells that are in *unknown\_cell* list. The probability of cells in the chain is by using (*reveal / unknown neighbors)* and the probability of cells in *unknown\_cell* list is by using *(all remaining mines / len(unknown\_cell))*. And for all of the probabilities, we always choose the cell with the least probability to be next step. Some important theorems and proofs we based on about probabilities of a cell have partly been claimed in the answer of Question 2 **STEP 0**.

**4**

**Chain of influence:**

1. **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?**

In our program, upper left corner is the first cell to visit. We get the neighbors of the current cell, after removing the neighbors whose situations are known (mined or clear), we put the rest into a set, which is one element of the chain. Then we explore another cell, get another set. These many sets form the chain of influence.

Each time we add a set into the chain, we traverse the entire chain. If there is a set that is a subset of another set, we can simplify the larger set. For example, if the upper left corner [0, 0] reveals number 1, there are one mine in [1, 0], [0, 1] and [1, 1] (Figure 4-1). Next step, if the computer goes to [0, 1] and reveal number 2, we remove the uncover neighbors, what’s left is two mines out of four surrounding cells (Figure 4-2). We traverse all sets, finds that ([1, 0], [1, 1]) is that subset ([1, 0], [1, 1], [1, 2], [0, 2]). Then we can simplify the current set to ([1, 2], [0, 2]), and this set contains 1 mine (2 – 1). Base on this simple idea, we can get an optimal chain of influence.

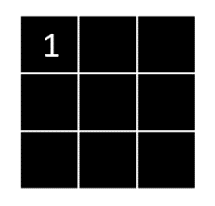
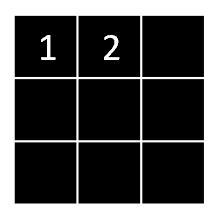
 

Figure 4-1 Figure 4-2

1. **What influences or controls the length of the longest chain of influence when solving a certain board?**

The way we choose the next step influences the length of the longest chain of influence. For starters, we use a priority queue to determine the next move. When no move is certainly safe, the safe probability determines the priority. For example, when the upper left corner [0, 0] reveals number “1”, the next step goes to [1, 1] and reveals number “2”. Each of cell [0, 0]’s uncover surrounding cells has a probability of 1/2 is mine while each of [1, 1]’s remaining neighbors has a probability of 1/5, then [1, 1]’s neighbors will be visited first, see figure 4-3. Because every time we get the simplest set, the use of priority may lead to some sets that have no subset in the chain, so theoretically the chain will become longer.

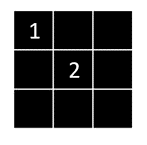
****

Figure 4-3

1. **How does the length of the chain of influence the efficiency of your solver?**

The longer the chain becomes, the less efficient the program will be, as it takes longer time to traverse through the chain.

1. **Experiment. Can you find a board that yields particularly long chains of influence? How does this vary with the total number of mines?**

As is stated in 2, when randomly pick up happens to pick up the diagonal cell, the chain tends to grow longer. But the length of the chain isn’t necessarily influenced by the number of mines. In a certain board, if the total number of mines is very large, the chain is usually shorter because there are usually more subsets that we can determine as mines, then less sets will be added to the chain; However, if the total number of mines is very small, we can also get a very small chain, for example, in an extreme circumstance, if there is only one mine in the lower right corner, we can always eliminate sets that are “irrelevant”, the chain will remain small. During the test, a “median” number of mines will get a relatively longer chain.

1. **Experiment. Spatially, how far can the influence of a given cell travel?**

During the test, the relationship between the number of mines and the length of the chain is Quadratic dependence, which is y = a \* (x – b) ^ 2 + c, (a < 0), where x is the number of mines, y is the length of chain. In a certain board, when the number of mines is b, c is the furthest the chain can go. Figure 4-4 is the relation between the number of mines and the length of chain of influence in a 4 \* 4 board.

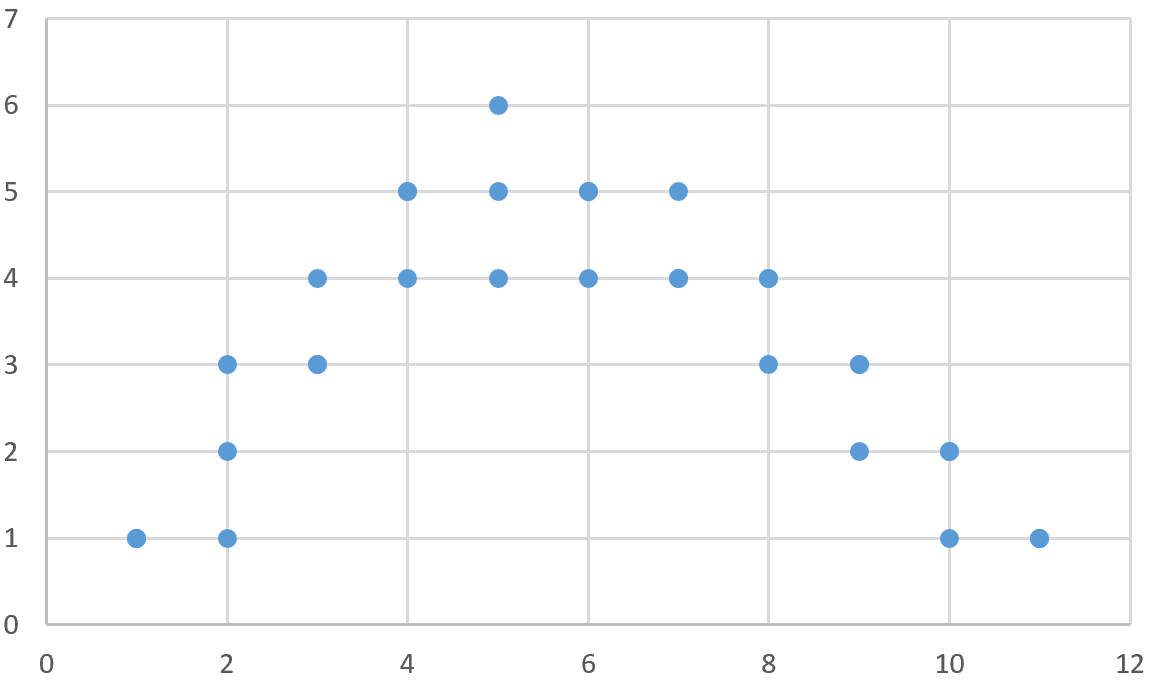


Figure 4-4

1. **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?**

As is stated in 3, the use of priority queue potentially grows the chain. When the decision is completely random and the probabilities are the same, we can pick the cell that is “next” to it. For example, if [0, 0] reveals a number “1”, we can program the computer to always choose [0, 1] or [1, 0]. This way can guarantee to get the set that has the subset in the chain.

1. **Is solving minesweeper hard?**

The basic idea of solving minesweeper is simple. It is a process of simplifying sets. In terms of space complexity, the use of space is linear. But in terms of time complexity, solving minesweeper is hard. Because one-time traversal is not enough to get the simplest chain. Each time we find a mine, the previous sets will potentially be influenced, in this way we need to traverse the chain again. Every step will take O(n^2) time to get the optimal chain of influence.

**5**

**Dealing with Uncertainty:**

1. **When a cell is selected to be uncovered, if the cell is ‘clear’ you only reveal a clue about the surrounding cells with some probability. In this case, the information you receive is accurate, but it is uncertain when you will receive the information.**

If we are not receiving any number from the current cell, only to know that it is clear, we will access the next safe cell or the relatively safer cell to continue the game. In the program, we use a variable *receive\_prob* to control its probability of not showing number. If its value is “1”, it always shows a number; If its value is “0.8”, it has 20% chances of not showing number.

1. **When a cell is selected to be uncovered, the revealed clue is less than or equal to the true number of surrounding mines (chosen uniformly at random). In this case, the clue has some probability of underestimating the number of surrounding mines. Clues are always optimistic.**

If the clue underestimates the number of surrounding mines, we can only trust the revealed number which equals to surrounding unknow cell, which means they are all mines. In other cases, we can infer nothing except adding these clues to chains of influence.

1. **When a cell is selected to be uncovered, the revealed clue is greater than or equal to the true number of surrounding mines (chosen uniformly at random). In this case, the clue has some probability of overestimating the number of surrounding mines. Clues are always cautious.**

Similar as Q2, we can only conclude that 0 means no mine surrounding. In other cases, we cannot infer there is a mine or not.