Type *Markdown* and LaTeX: α^2

The solution for CS520 spring 2019 assignment 2

Team Members: Wenhao. Luo, Renfeng. Xu, Bo. Mao

Email: w.luo.bosonfields@rutgers.edu (mailto:w.luo.bosonfields@rutgers.edu)

Project URL: https://github.com/bosonfields/mineSweeper.git (https://github.com/bosonfields/mineSweeper.git)

Section_1: Background: MineSweeper

Description for the file we uploaded:

- The relevant code files are packaged in the ***cs520_Ass1_Wenhao Luo***
- The project is also uploaded on the public github repository, see the ***Project URL***
- The file ***minesweeper.py*** is the ***GUI*** class and the main function
- The file ***mineblock.py*** is the board class in which to generate the ***board*** of the game and some function to read the information in the board
- The file ***agent.py*** is the AI robot to play the game

Section_2:Program Specification

Issue(i): The environment should take a dimension d and a number of mines n and generate a random d × d boards containing n mines. The agent will not have direct access to this information.

Sol:

- We implemented a board (mineblock.py) and we built a board with $Width \times Height$ boards containing n mines.
- For the first step, we make some tricks to make sure that the first step for the agent is not a mine.

Issue(ii): In every round, the agent should assess its knowledge base, and decide what cell in the environment to query.

Sol:

- We built a list of frontier to contain the node in which there is a board with $Width \times Height$ boards containing n mines.
- We also built a function in the board which is called double_check, when the agent call that it will get the information of the tiles around it.

Issue(iii): In responding to a query, the environment (or user) should specify whether or not there was a mine there, and if not, how many surrounding cells have mines.

Sol:

- When the agent use the tiles already in the frontier, for each tile it will call the funciton double check to get the information around it, the specific details are showed in below:

```
  def double check(node):
       return no_click_tiles # Tiles that are not clicked around the node
              all_need_open # Boolen value to determine whether all around tiles
need be opened
              all_need_flag # Boolen value to determine whether all around tiles
need be flaged
              remain_mines_around # The number of mines that around the node not
determined
```

Issue(iv): The agent should take this clue, add it to its knowledge base, and perform any relevant inference or deductions to learn more about the environment. If the agent is able to determine that a cell has a mine, it should flag or mark it, and never query that cell

Sol:

- We later built a function solver to implement the knowledge base that agent AI gotted
- We can see the two Boolen value above, the ***all_need_open*** and the ***all_need_flag***, by using these two values, our AI can do some basic choices
- As the results and code shows, initially, for each tile, the agent will open or flag all the remain un-open tiles around it when the number of remain mines is zero or the number of remain tiles equal to the information in the knowledge base

Issue(v): Traditionally, the game ends whenever the agent queries a cell with a mine in it - a final score being assessed in terms of number of mines safely identified.

Sol:

- In our interfaces, the un-flaged mines(those bombed) is showed on the right side and we will use this value to grade the result

Issue(vi): However, extend your agent in the following way: if it queries a mine cell, the mine goes off, but the agent can continue, using the fact that a mine was discovered there to update its knowledge base. In this way the game can continue until the entire board is revealed - a final score being assessed in terms of number of mines safely identified out of the total number of mines.

Sol:

- Those bombed mines are also considered in our two boolen values: the ***all_need_open*** and the ***all_need_flag***. The detailed double check function are listed below

```
def double_check(self, x, y):

     Iterating all the node can be influence by tile (x, y)
          return sum_flaged # The number of tiles that are flaged
                 sum_bombed # The number of tiles that are bombed
                 sum_no_click # The number of tiles that are not opened
                 around_count # The number of mines around the tile (x, y)

     all_need_flag = around_count - sum_flaged - sum_bombed == sum_no_click
     all_need_open = around_count - sum_flaged - sum_bombed == 0
     remain_mines_around = around_count - sum_flaged - sum_bombed
```

- The final score of the AI will be calculated based on the situation, bomb cells are learned by the agen AI

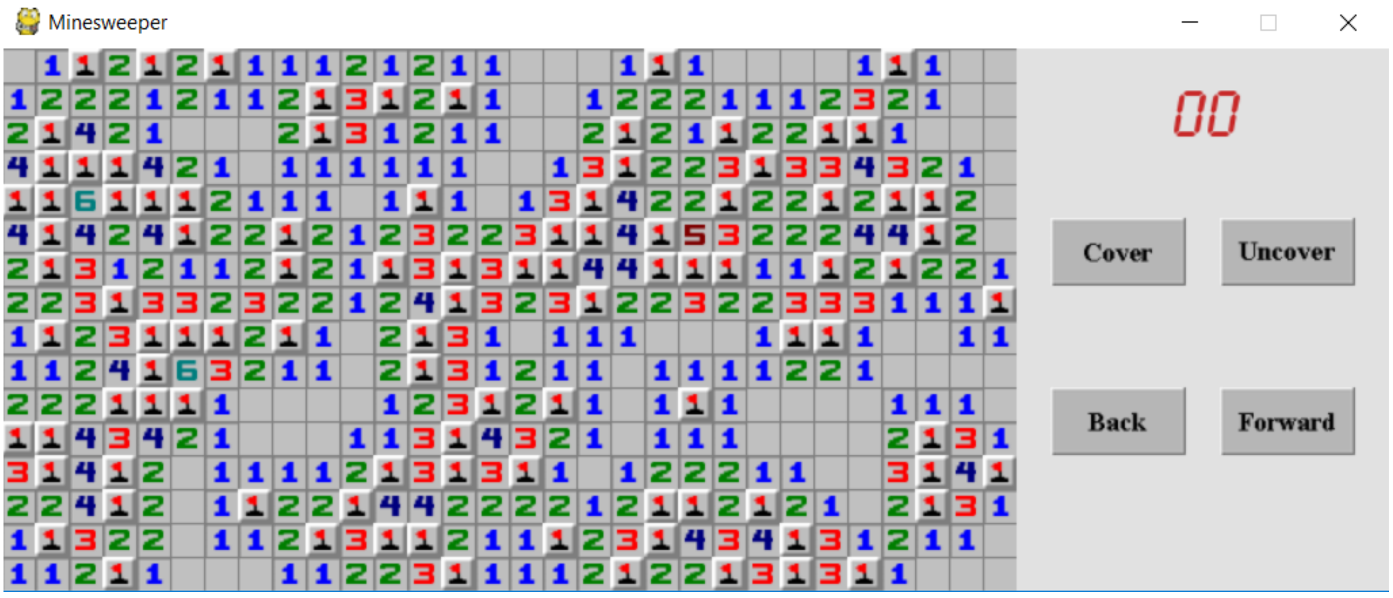

Figure 1. This is the interfaces of our program, the basic images of button was downloaded from google or made by myself. The button of back and forward can be used to trace the each steps of our AI robot

Section_4

Issue(i): 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?

Sol:

- Just as we have mentioned before, for every step of our AI, we will check the basic statement, that is the decision we can make based on the single tile in our knowledge base.
- So, only when the information or the tile in our frontier cannot singly help our AI to make the definite decision, we will use the chain of influence
- In this case, intuitively all tiles in the frontier will be considered into the chain of Influence
- Let me give an example to show the data structure of our chain of influence:

Table : An example for chain of influence

A	B	C	D	E
(0,1): 1	(1,1): 2	(2,1): 1	(3,1): 2	(4,1): 1

- As the Table showed above, the first line is the covered tiles near the frontier and the second line is the knowledge base, that is the coordinates of the tile and the information it supported
- The data structure for the knowledge base is ***Dictionary { List (coordinate): Int (number of surranding mines) }***
- Then, we change the knowledge base into another form, for example considering the the ***cell (1,1), (2,1) and (3,1)***. The final form released as below

$$(1,1) => \{(A,B,C) \longrightarrow [(A,B),(B,C),(A,C)]\}$$

$$(2,1) => \{(B,C,D) \longrightarrow [(B),(C),(D)]\}$$

$$(3,1) => \{(C,D,E) \longrightarrow [(C,D),(D,E),(C,E)]\}$$

- In the new dictionary { key : value }, the key is the ***un-opened tiles*** that we considered together, the value is the ***combinations for all the possiblity***. In this case, each combination in the value is one possible result that all node in this result are mines
- Now, let me us combine the information of (1,1) and (2,1). That is using ***Cartesian Product*** to combine all the possible solutions in node (1,1) and (2,1) and then check each possiblity with the information we already have

	(A,B)	**(B,C)**	**(A,C)**
(B)	(A,B)	False	False

	(A,B)	(B,C)	(A,C)
(C)	False	False	(A,C)
(D)	False	False	False

- Then we can get the new chain of influence:

$$(A, B, C) \cup (B, C, D) = (A, B, C, D)$$
$$\{(A, B, C, D) : [(A, C), (A, B)]\}$$

- We can use this strategy to combine this information with the next dictionary until we get the all posibilities
- At the end, we will get the full possibilities. ***Note that, in this case we only collect the nodes are mines in coding we need also build a dictionary to collect nodes that are normal***
- Finally, for all possiblities, we will calculate the intersection of all the possibilities in the list. If the intersection is not empty, then we will know that all the nodes in the last intersection set must have mines.
- For example, let us choose the chain of (A,B,C,D), which we showed before as an illustration, we intersect all the possiblities, that is (A,C) and (A,B) in this case. After all the intersect calculation, we will get a set (A). So we can flag the tile A definitely, since for all possibilities, node A is always a mine.
- This calculation can work for a long chain, for example, we have a long chain of influence:

$$\{(A, B, C, D, E, F, G) : [(B, C, E, F)(A, B, C)(B, C, D, G)(A, B, C, G)]\}$$

$$(B, C, E, F) \cap (A, B, C) \cap (B, C, D, G) \cap (A, B, C, G) = (B, C)$$

- Then we can mark the flag on B and C. For the blank situation, this strategy can also be applied in the same way.

Issue(ii): What influences or controls the length of the longest chain of influence when solving a certain board?

Sol:

- The longest chains of influence is highly influenced by the density of the mines in the game, since the more mines in the game, the less consecutive clues we will get. If clues cannot be consecutive, the probability to get a result is less
- However, when the density is too high, the longest chain of influence will reduce when the number of mine reaches some level. We can see that in firgure below. That is because our AI agent is frequently blocked by the mines, so the chains cannot be long. For example, when the mine density is 0.7, the longest chain can reduce to 3 or 2 as usual since our search is frequently blocked by the mine.

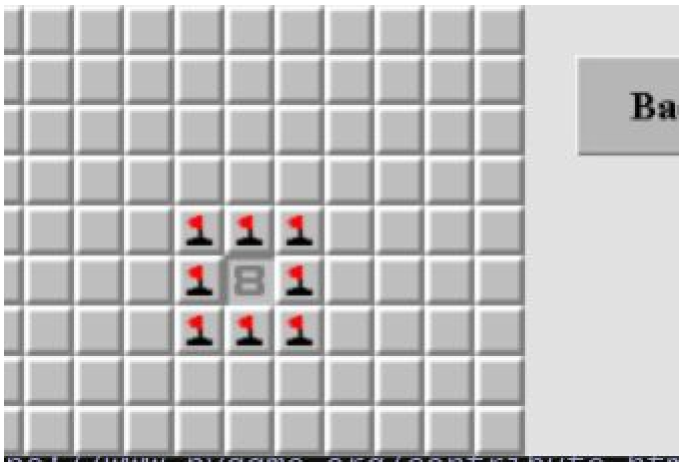

figure 2. In this figure, we can see that our detection is blocked, in this case our chains of influences cannot be long.

Issue(iii): How does the length of the chain of influence influence the efficiency of your solver?

Sol:

- When the mine's density is not very high (less than 0.5), and only implement the game with the simple solution (that is ***all_need_flag*** or the ***all_need_open*** is true, otherwise make a random choice), the smaller the score we will get.
- However, since we modified our AI robot with the chain of influences showed above, the longest chain have little influence on our AI robot

Issue(iv): Experiment. Can you find a board that yields particularly long chains of influence? How does this vary with the total number of mines?

Sol:

- As we mentioned before, the the longest chains of influences is highly influenced by the mine density, just see the ***figure 3*** , ***the longest chain is usually find out at the density of 0.3***. According to the simulation, the maximum value is at density of 0.35. That is the density is high enough, so we can hardly have consecutive nodes to analysis, but the not too high so our chains of influence will not be blocked, just like the ***figure2*** revealed

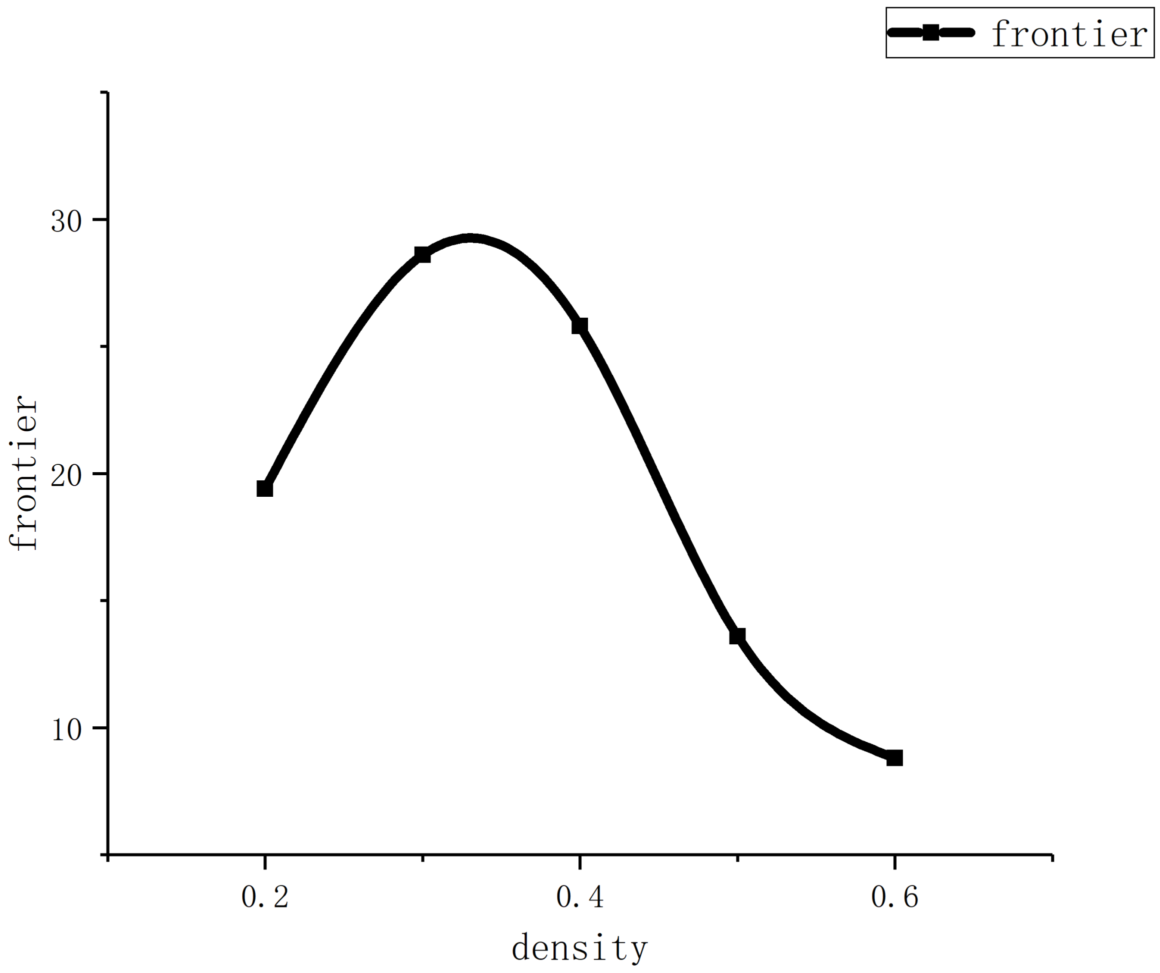

Figure 3. The length of langest chains of influences(frontier) versus the mine density. We get the maximum value at 0.35

Issue(v): Experiment. Spatially, how far can the influence of a given cell travel?

Sol:

- We runned several code here to check the longest influence chain. It is usually that the longest chain is about 3 to 4 node of information and can influence about 5 to 6 un-opened node. See the figure for an example

A	B	C	D	E
F	G	(1,1): 2	H	I
(0,2): 1	(1,2): 2	(2,2): 1	(3,2): 2	(4,2): 1

- In this particular example, the given cell of (4,2) can influence as long as to (1,2), which can influence 5 un-opened tiles. We later use such patter to modify our algorithm

Issue(vi): 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?

Sol:

- We implemented two algorithm to modify the simple algorithm. ***One is based on the chains of influences of section 4, we combinate consecutive chains of influences, each combination combines 4 chains which means we can have as usual 5 un-opended tiles to determine. And this is about*** $2^5 = 32$ ***possiblities for each combined chains.*** Then we later check the condition we have and get the final intersections of each possiblity to determine whether we can make a obsolately true step instead of guess.
- Another algorithm we use is to use the patterns we find in the game to make the algorithm become more efficient

Algorithm based on combination of chains of influences

```
    def force_crack_prepare(self, frontier, MineBlock, influence_len):
        Dict_inf = {}
        inf = []
        Comb_inf =[]
        if frontier:
            for pair in frontier:
                x, y = pair
                around, Allempty, Allmine, around_count = MineBlock.double_check(x
, y)
                if around:
                    Dict_inf[around] = around_count
                    inf.append(around)
            for node in inf:
                i = inf.index(node)
                if i+influence_len<len(inf):
                    tmp = copy.deepcopy(inf[i:i+influence_len])
                else:
                    tmp = copy.deepcopy(inf[i:])
                Comb_inf.append(tmp)
        return Dict_inf, Comb_inf
    def force_crack(self, Dict_inf, Comb_inf):
        for five_list in Comb_inf:
            combine = set()
            for influence in five_list:
                combine = combine | set(influence)
            combine_list = list(combine)
            num = len(combine_list)
            Dict_value = {}
            for node in combine_list:
                Dict_value[node] = 0
            possible = []
            for i in range(2**num):
                binary = list(str(bin(i))[2:])
                for node in reversed(combine_list):
                    Dict_value[node] = int(binary.pop())
                if self.condition_figure(Dict_value, five_list, Dict_inf):
                    one_possible = []
                    for node in Dict_value:
                        if Dict_value[node]:
                            one_possible.append(node)
                    possible.append(one_possible)
            inter = set(possible.pop())
            for poss in possible:
                inter = inter & set(poss)
            if inter:
                return inter
    def condition_figure(self, Dict_value, five_list, Dict_inf):
        cond = True
        for influence in five_list:
            sum = 0
            for node in influence:
                sum = sum + Dict_value[node]
        if sum != Dict_inf[influence]:
            cond = False
            break
    return cond
```

Algorithm based on my pattern determination

```
def pattern_determine(self, frontier, MineBlock):
        List32 = []
        List31 = []
        List21 = []
        ListN1 = []
        find = False
        if frontier:
            count = len(frontier)
            for pair in frontier:
                x, y = pair
                around, Allempty, Allflag, current_count = MineBlock.double_check(
x, y)
                if around:
                    if len(around) == 3 and current_count == 2:
                        List32.append(set(around))
                    elif len(around) ==3 and current_count ==1:
                        List31.append(set(around))
                    elif len(around) == 2 and current_count ==1:
                        List21.append(set(around))
                    elif current_count ==1:
                        ListN1.append(set(around))

            for two_mine_tuple in List32:
                for one_mine_tuple in List31:
                    inter_1 = two_mine_tuple & one_mine_tuple
                    if inter_1 and len(inter_1) == 2:
                        item_mine = (two_mine_tuple - inter_1).pop()
                        MineBlock.flag_node(item_mine)
                        self.flag_record(item_mine)

                        item_normal = (one_mine_tuple - inter_1).pop()
                        MineBlock.open_node(item_normal)
                        self.no_mine_record(item_normal)
                        frontier.append(item_normal)

                        List32.remove(two_mine_tuple)
                        List31.remove(one_mine_tuple)
                        List21.append(inter_1)
                        break

            for three_tuple in List31:
                for two_tuple in List21:
                    inter_2 = three_tuple & two_tuple
                    if inter_2 and len(inter_2) == 2:
                        item_normal = (three_tuple - inter_2).pop()
                        MineBlock.open_node(item_normal)
                        self.no_mine_record(item_normal)
                        frontier.append(item_normal)

                        List31.remove(three_tuple)
                        break
```

```python
            for three_two_tuple in List32:
                for two_tuple in List21:
                    inter_3 = three_two_tuple & two_tuple
                    if inter_3 and len(inter_3) == 2:
                        item_mine = (three_two_tuple - inter_3).pop()
                        MineBlock.flag_node(item_mine)
                        self.flag_record(item_mine)

                        List32.remove(three_two_tuple)
                        break

            for N_tuple in ListN1:
                for two_tuple in List21:
                    inter_4 = N_tuple & two_tuple
                    if inter_4 == two_tuple:
                        item_empty = list(N_tuple - inter_4)
                        for node in item_empty:
                            MineBlock.open_node(node)
                            self.no_mine_record(node)
                            frontier.append(node)

                        ListN1.remove(N_tuple)
                        break

            if len(frontier) - count:
                find = True
return find
```

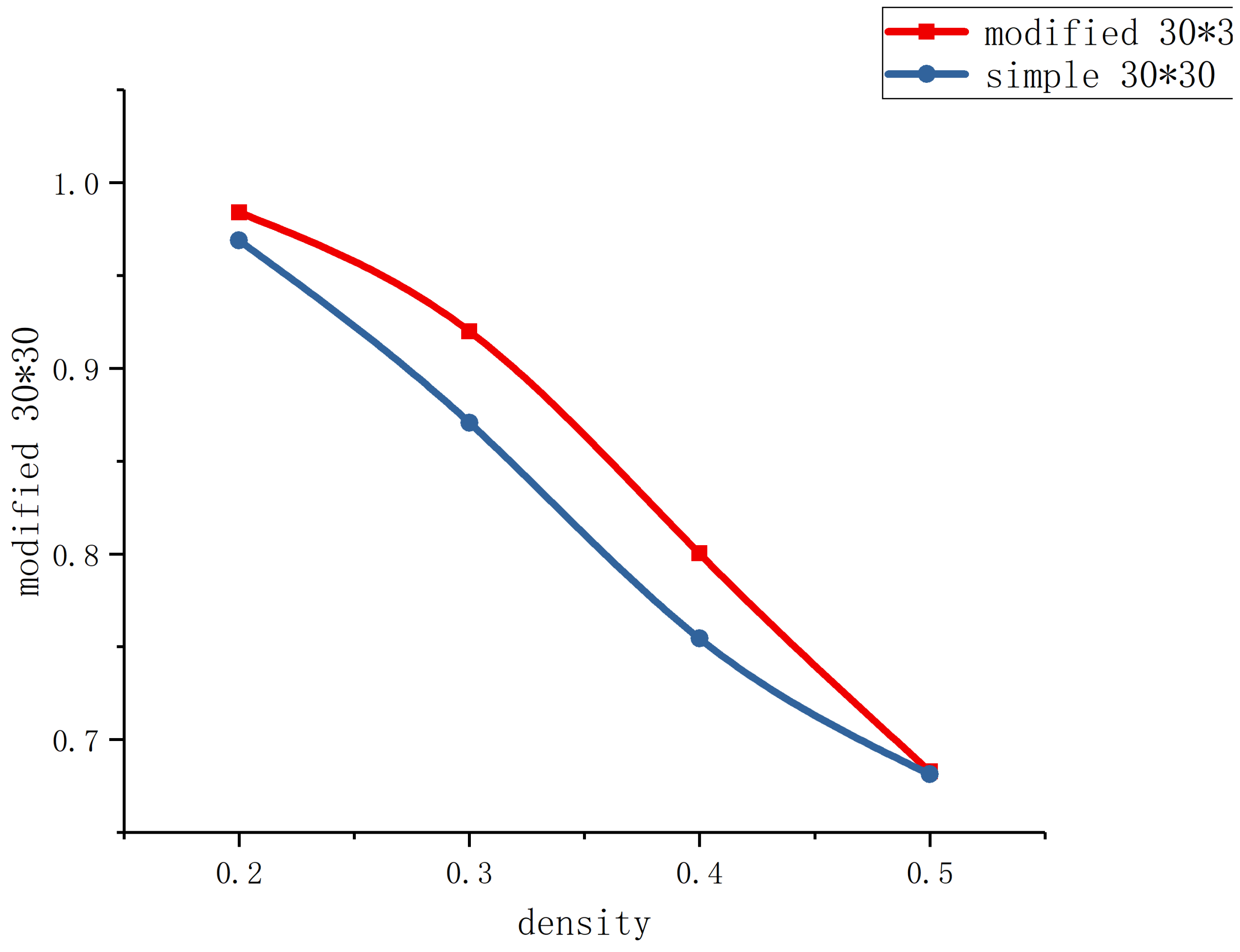

Figure 4. The comparison between the two algorithm: the blue one that is only for simple determination and guess, the red one is for the modified algorithm with both the algorithm based on pattern detemination or chains of influence combination.

- Both two algorithm works pretty well, you can see the ***figure 4*** that, when the mine density less than 0.6, our modified algorithm works much better.

Issue(vii): Is solving minesweeper hard?

Sol:

- I think our algorithm works pretty well.

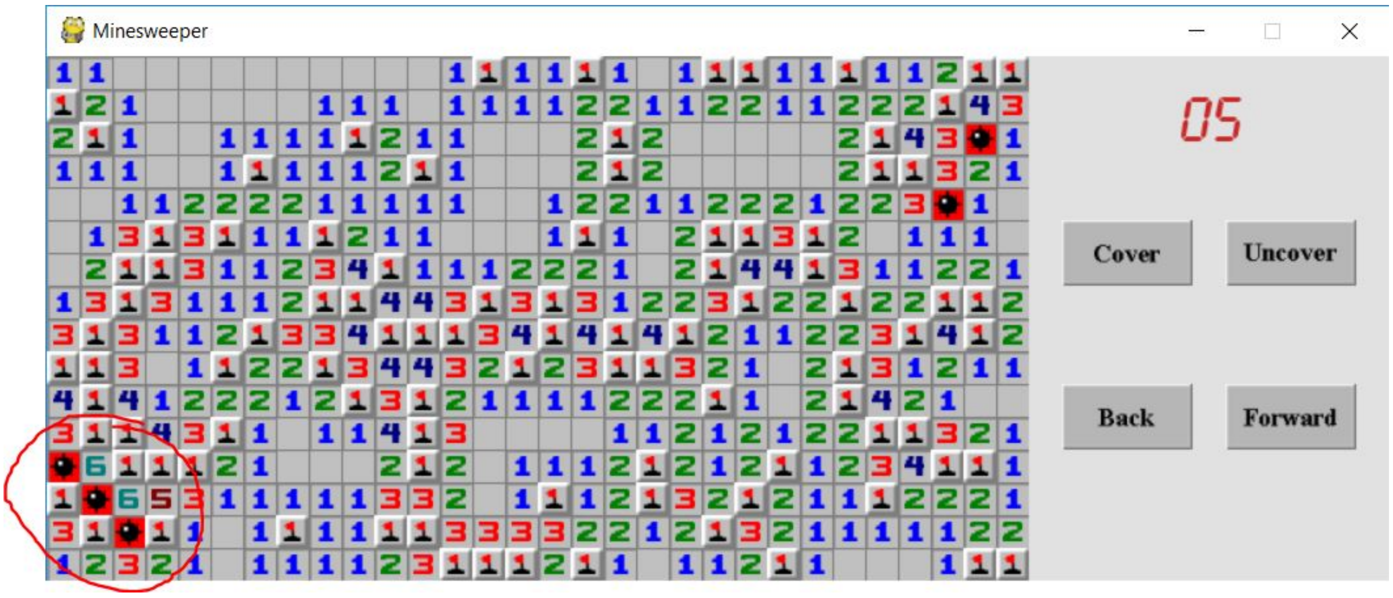

figure 5. We can see that this work not done well (this is a standard minesweeper), this is because the first step is not good, with less information, our agent made a lot of random guess, so it made a lot of errors

- However, no matter how good your algorithm is, they can still make a lot of mistakes. Especially the first step, in figure 5, our agent start at the left down angle, it made a lot of guess. The result is three mistakes, we cannot reduce such mistakes