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

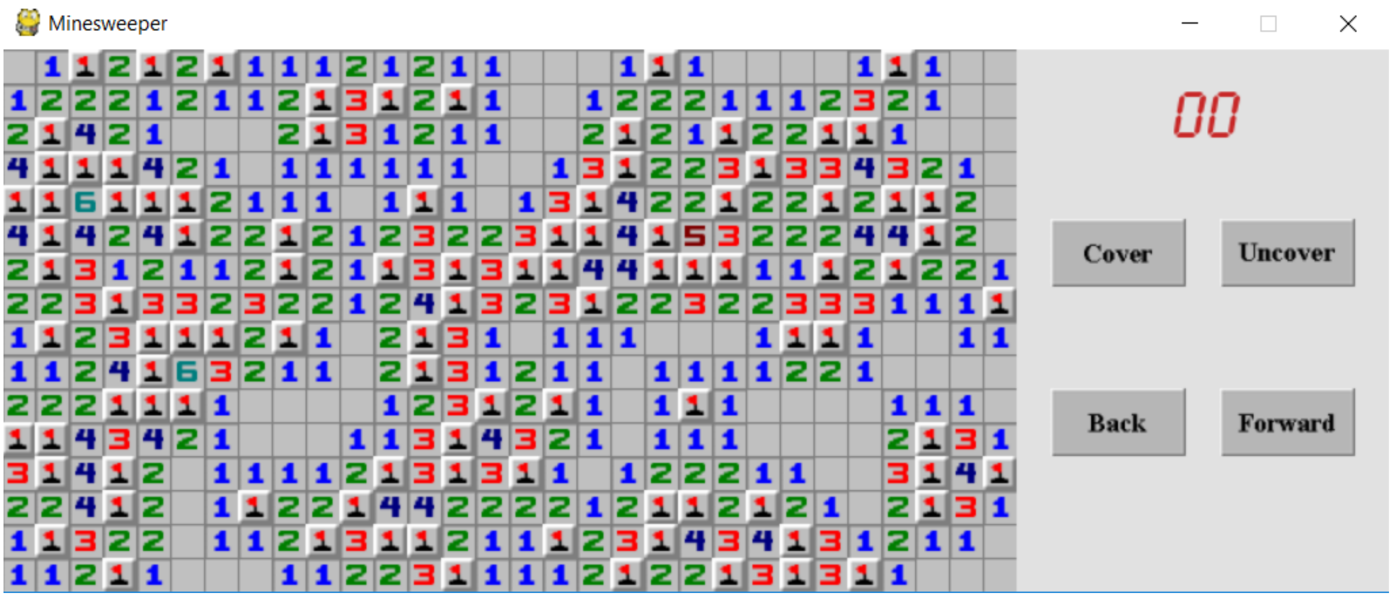
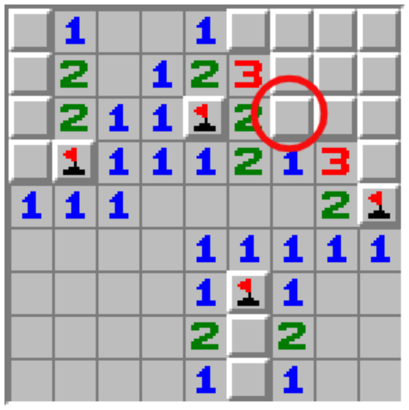


Figure 1. This is the interfaces of our program, the basic images of button was downloaded from google or made by myself. The button of back and forward can be used to trace the each steps of our AI robot

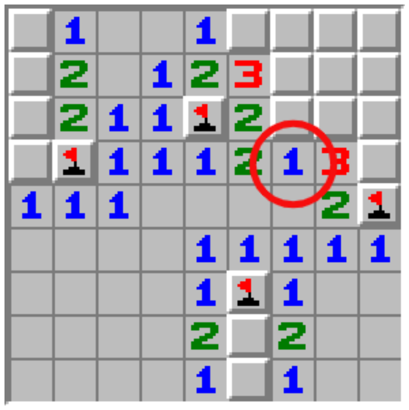
section 3

3.1 representation:

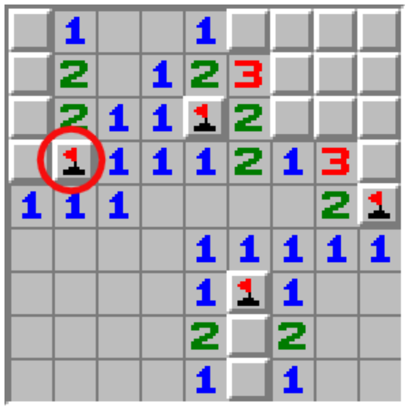
- Map: A simple n*m numerical array representing the distribution of the mines. On every position, -1 represents having a mine, 0 represents not having mines.
- Board: A n*m array in which the elements are of class Block,
  - `Block.x;Block.y` represent its coordinates
  - `Block.value` : -1 when there is a mine, i ∈ (0,...,8) when no mine but has i mines surrounding
  - `Block.status` :
    - `'unsearch'` : Agent hasn't uncover this block



    - `'no_mine'` : Agent has uncovered this block

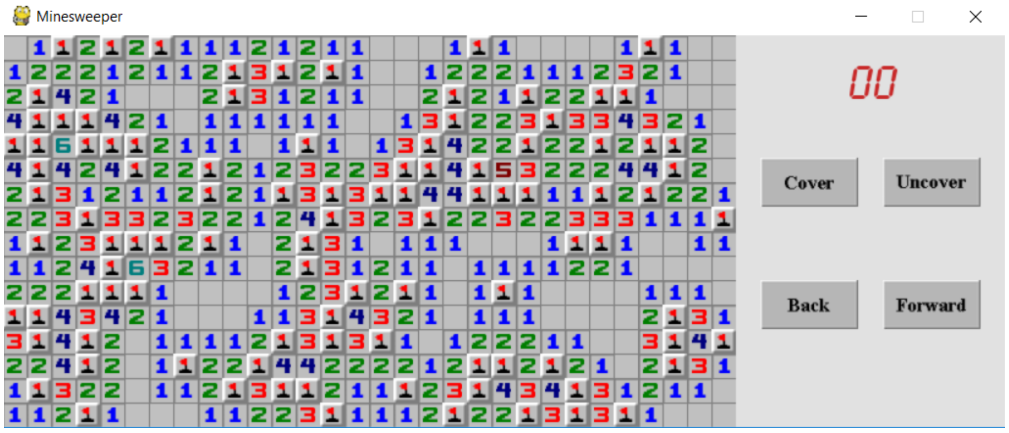


    - `'flag'` : Agent has predicated that there is a mine



```python
class Block(object):
    def __init__(self,x,y,value=0,status=''):
        self.status= status
        self.x = x
        self.y = y
        self.value = value
```

- GUI: we use pygame to display every step of Agent until the end of the game. Besides we can also trace back the process of Agent playing the minesweeper step by step.



- Knowledgebase:
  - Basic information: besides from the information stored in Board, we use several lists to store different groups of block
    - `Flaglist = []` , which contains the blocks which are plugged a flag by the Agent.
    - `Securelist = []` . which contains blocks of which the surroundings are all clear. And the block itself is no need of looking at any more
    - `Frontier = []` , which contains blocks which are clear but its surroundings are still unclear.
  - Logical propositions: for each block whose surrounding is uncertain, we create an instance of `class Knowledge` , using attributes `space` and `mines` denoting the unsearched spaces and how many mines are in those spaces. Every instance represents a proposition generated from the game. For example `Knowledge([(1,2),(1,1)],1)` represents a knowledge denoting in (1,2) and (1,1) exists one mine

```
class Knowledge(object):
    def __init__(self,point=[],number=0):
        self.point=point
        self.number=number
    def judge(self,dic):
        sum=0
        for i in range(len(self.point)):
            sum=sum+dic[str(self.point[i])]

        if sum==self.number:
            return True
        else:
            return False
```

3.2 Inference

- Simple Inference:
    - If the Agent is standing on a block where the number of unsearched blocks equals the number of mines need to be swept around it. The agent just simply plug a flag on every unsearched block and change the `block.status` to `'flag'`
    - If the Agent is standing on a block where the number of flags around it equals to its value, meaning there is no more mines around. The Agent just simply clicks on the unsearched block around it. since the unsearched blocks around the Agent are all marked, the block where Agent stands are all done and no need to look at anymore, therefore we remove it from frontier to securelist.

```
    def simple_flag(self,value,mine,unsearchlist):

      if len(unsearchlist) == value - mine:
          for block in unsearchlist:
              block.status = 'flag'
              self.flaglist.append((block.x,block.y))
          self.securelist.append((self.x,self.y))
          self.watchlist.remove((self.x,self.y))

    def simple_click(self,value,mine,unsearchlist):

      if value==mine:
          for block in unsearchlist:
              block.status = 'nomine'
              self.watchlist.append((block.x,block.y))
              self.securelist.append((self.x,self.y))
              self.watchlist.remove((self.x,self.y))
              if block.value == 0:
                  self.x = block.x
                  self.y = block.y
                  self.securelist.append((self.x,self.y))
                  self.infinite_expand()
```

- Complex Inference:
  - The Agent tries to determine information of uncovered blocks by calculating the propositions in the knowledge base which contains those blocks. For example A,B,C,D,E represent blocks, in A,B,C there exists 1 mine; in B,D,E there exists 2 mines, in C,D,E there exists 2 mine. We can converge information from logical assignment to equations:

$$(A \land \neg B \land \neg C) \lor (\neg A \land B \land \neg C) \lor (\neg A \land \neg B \land C) <==> A + B + C = 1$$

    the knowledge set becomes

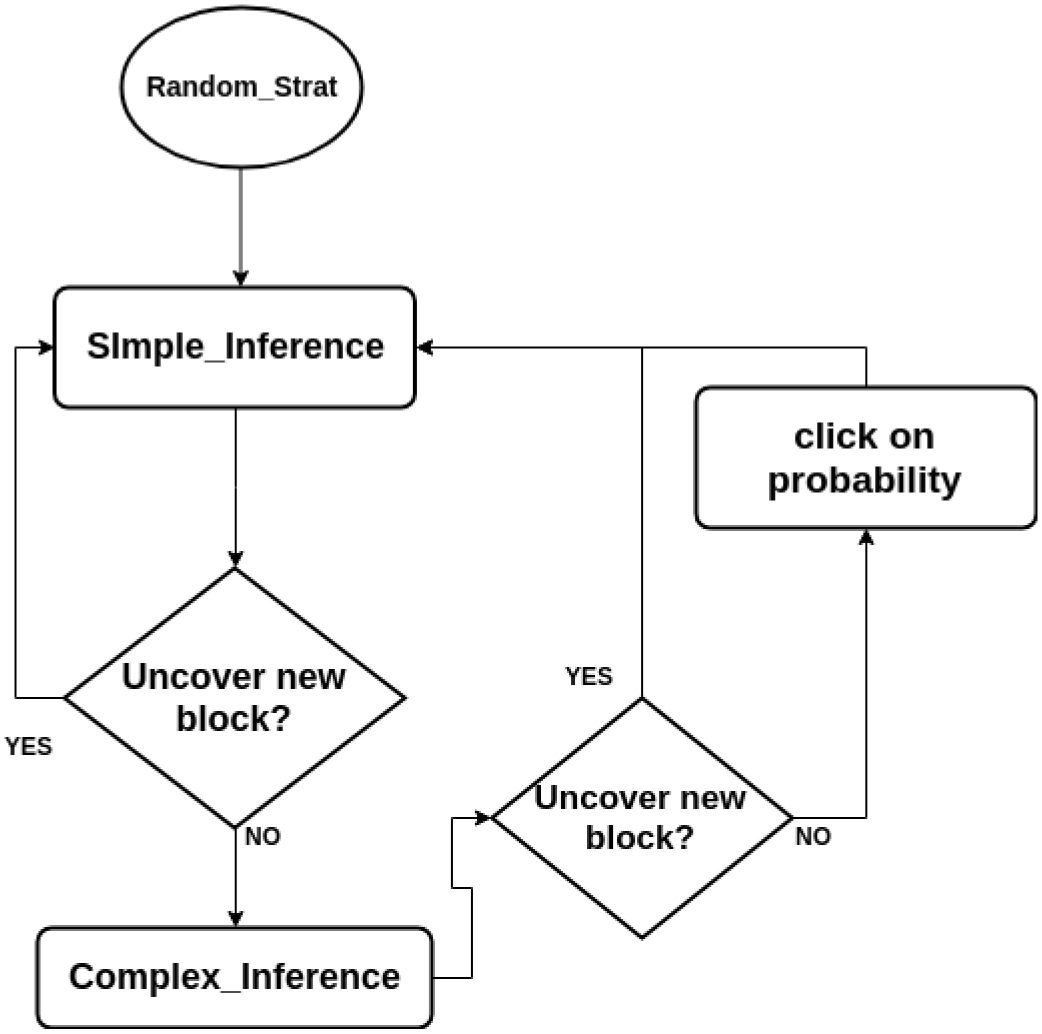
$$A + B + C = 1, B + D + E = 2, C + D + E = 2$$

    then solve the equation set, see if there is a unique solution for each block. Since all of the steps are processed by rigorous inference, so it is sure the decision of Agent is correct.

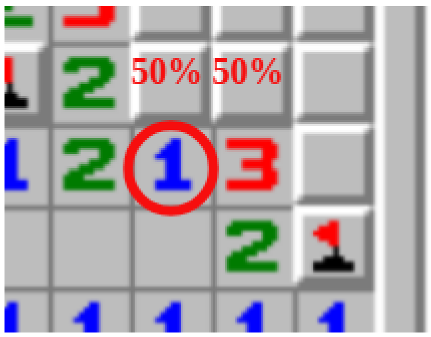
(Details are revealed in Section 4)

3.3 Decision

- the process of decision making :

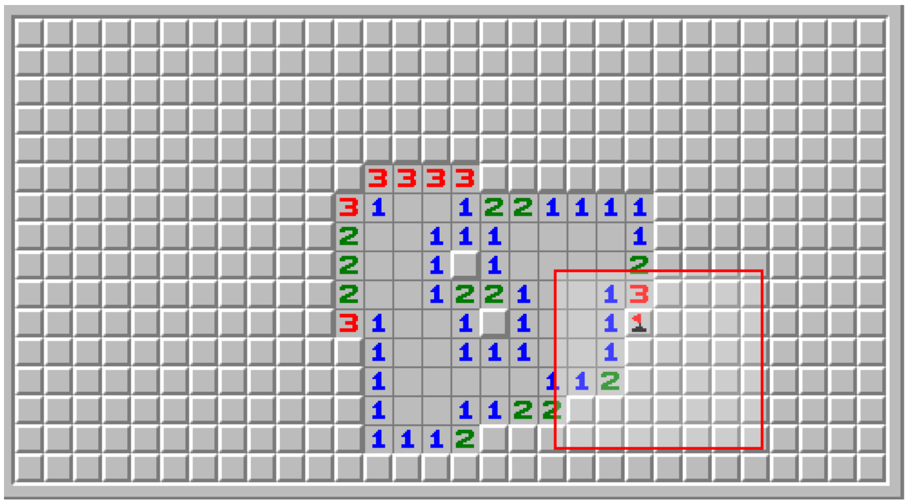


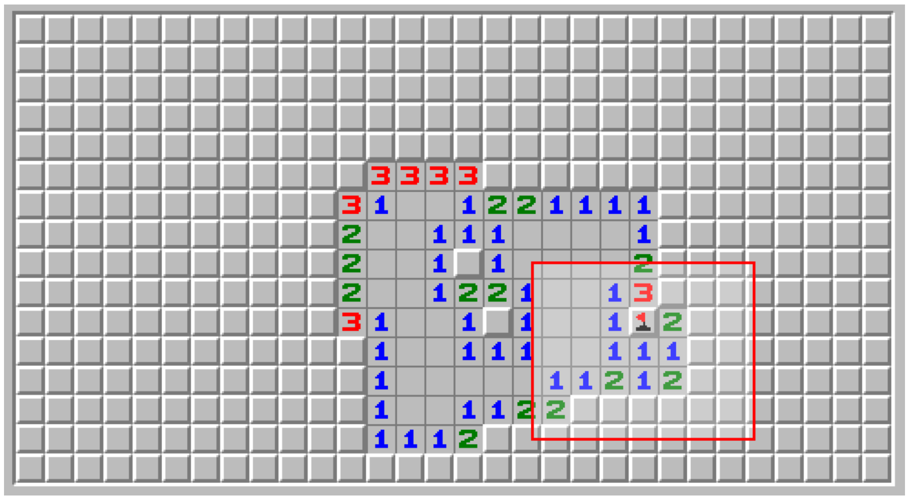
- Our Agent first processes the game with simple inference, because it is the most efficient method because it has a huge chain reaction on block around it and it works at most of the times.[*Chain reaction here means expanding one block has high poosibility of making its neighbors solvable] Then if there is no new reveals appearing, our Agent will try to use Complex Inference to logically determine the blocks, if there are new blocks revealed after it, we use simple inference again.
  - If no new information can be obtained by all of our current knowledge base. Then the only choice for the Agent is to take a risk and make a choice on those unsearched blocks. For every local assignment in our knowledge base, we can calculate the probability of unsearched block having mine. Then we accumulate the probability for one block from other assignments which contain it. And click on the block which has the least probability of having mine.

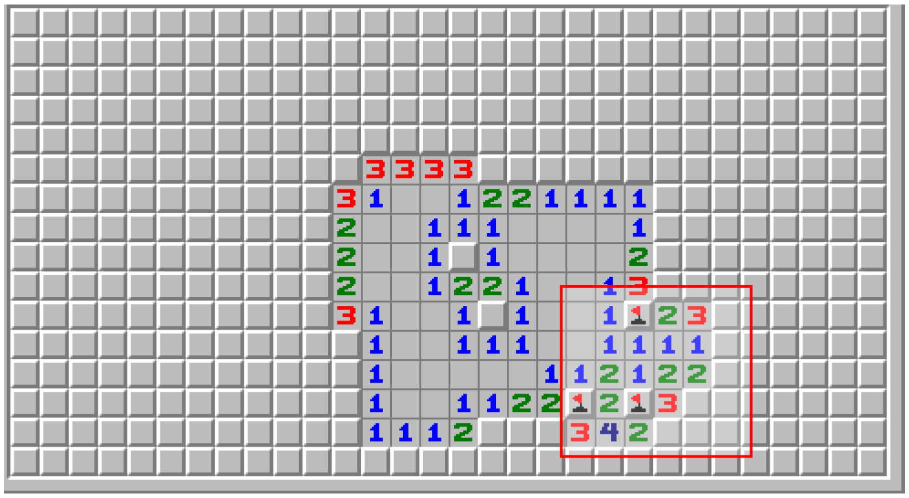


3.4 Performance(AI vs human)

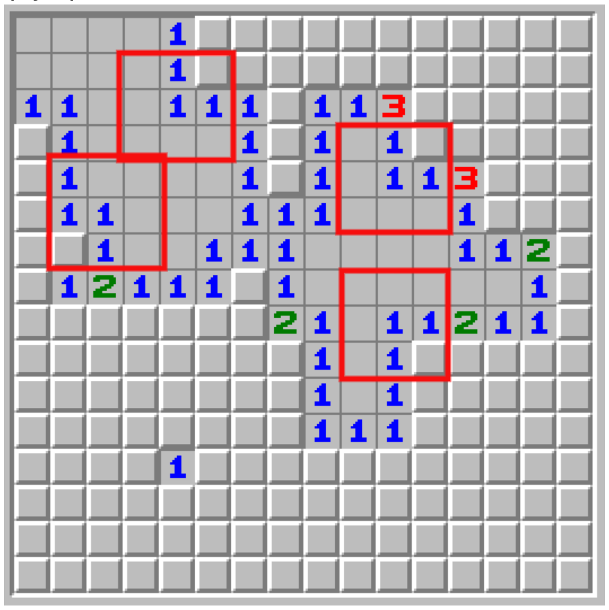
- Generally, our AI agent solves the mine in a very efficient, logical and human-like process. Because in the algorithms, every decision and predication is based on logical analysis, so every step of it is reasonable(except when Agent has to take a risk to keep the game process). Moreover, the logical flow of making the decision basically comes from how human play the minesweeper game. We follow the same process when playing it on our own. But there are still some differences because human not only plays by rules, we also play by experience.
    - common traversal vs pattern-using
        - Either in the Simple Inference process or Complex Inference process, our Agent does the traversal in a non-intelligent way, it simply traverses in the order of how data is stored in the data structure. For example, in Simple Inference, when a new block is revealed, its neighbor blocks are likely to be affected and then become solvable. But in this circumstance, our agent will traverse the `watchlist` once again, starting from `watchlist[0]` which might be at a remote position and has zero relationship with the previous predication. But this could be improved by combining DFS algorithm and our traversal algorithm, giving it higher priority to expand neighbors of a recent expanded block.







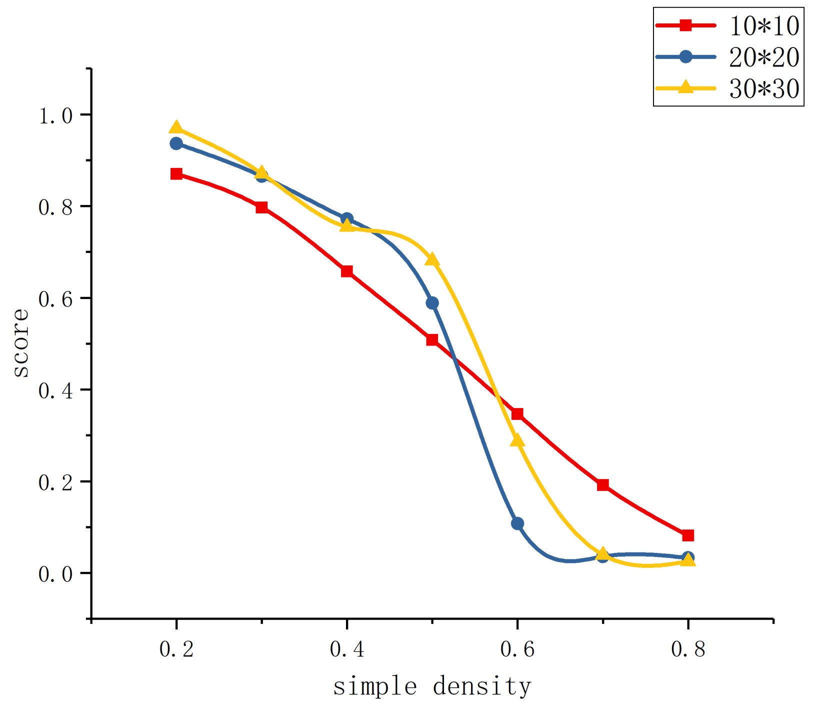
- Pattern Using: when we are playing the minesweeper, we tend to refer to our previous experience, therefore, not simply predicate by logic, but predicate by patterns. For example, it is very efficient to first look for the pattern below and put the flag on the empty space.



Because of the influence chain,we can expand the block very fast without wasting time processing other blocks.

3.5 Performance(mine density)

- By experimenting, we found that the higher density of mine, the lower score the agent will get, the reason are simple to think about. With very few mine, the secure areas are tend to adjacent with each other, making the chain of influence longer and making the solving faster. But when density get higher, the secure area become more likely to be intersected by the mine and therefore make the shut down the chain reaction. Our Agent has to take a risk multiple times, therefore more likely to make an error. When density approaches $[0.6, 0.7]$, we consider it every hard to solve a minesweeper.



3.6 Efficiency

- Our Agent never run into some time or space constraints in testing because we make sure it does not do duplicate traversal or recursion. But since our board size is limited, it can be solved in a short time. But if the board size gets relatively large. Time and space issues are worth taking care of.
    - Traversal takes time: Doing simple inference on blocks which does not reveal any new block is a huge waste of time, to improve it, we can either combine DFS or add priority on the traversal list.
    - Solving logical assignments: In complex inference, we generate knowledge based on the status of the game and solve that knowledge (assignment) to predicate new blocks and new information .but if we solve lots of composite assignment at the same time. It will take ridiculous of space and time especially the boards get big and `watchlist` gets long. However, The farther blocks are, the less their assignment are related. Add less related assignment does not really help solve it. Therefore controlling the size of the influence chain is a key, especially when board size gets big and there is more assignment.
      (details are revealed in section 4)

3.7 Improvements

- the influence of the number of mines is very less at the beginning or in between. It's effect gradually shows when there are few empty spaces left on the board. Then it will be helpful when solving knowledge base(assignment) and predicate the distribution of the last few mines.

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.

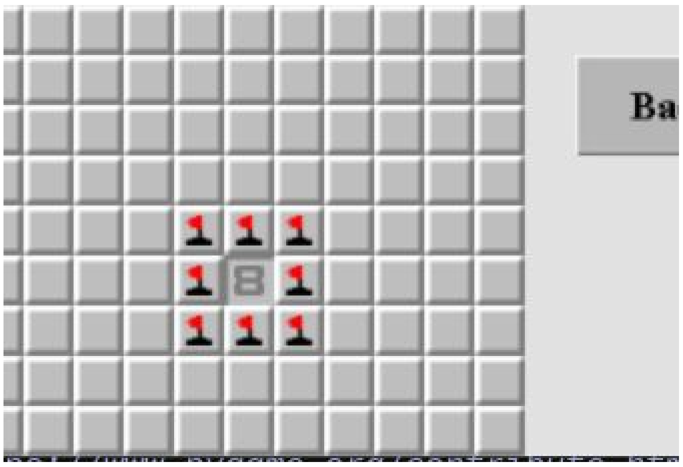


figure 2. In this figure, we can see that our detection is blocked, in this case our chains of influences cannot be long.

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

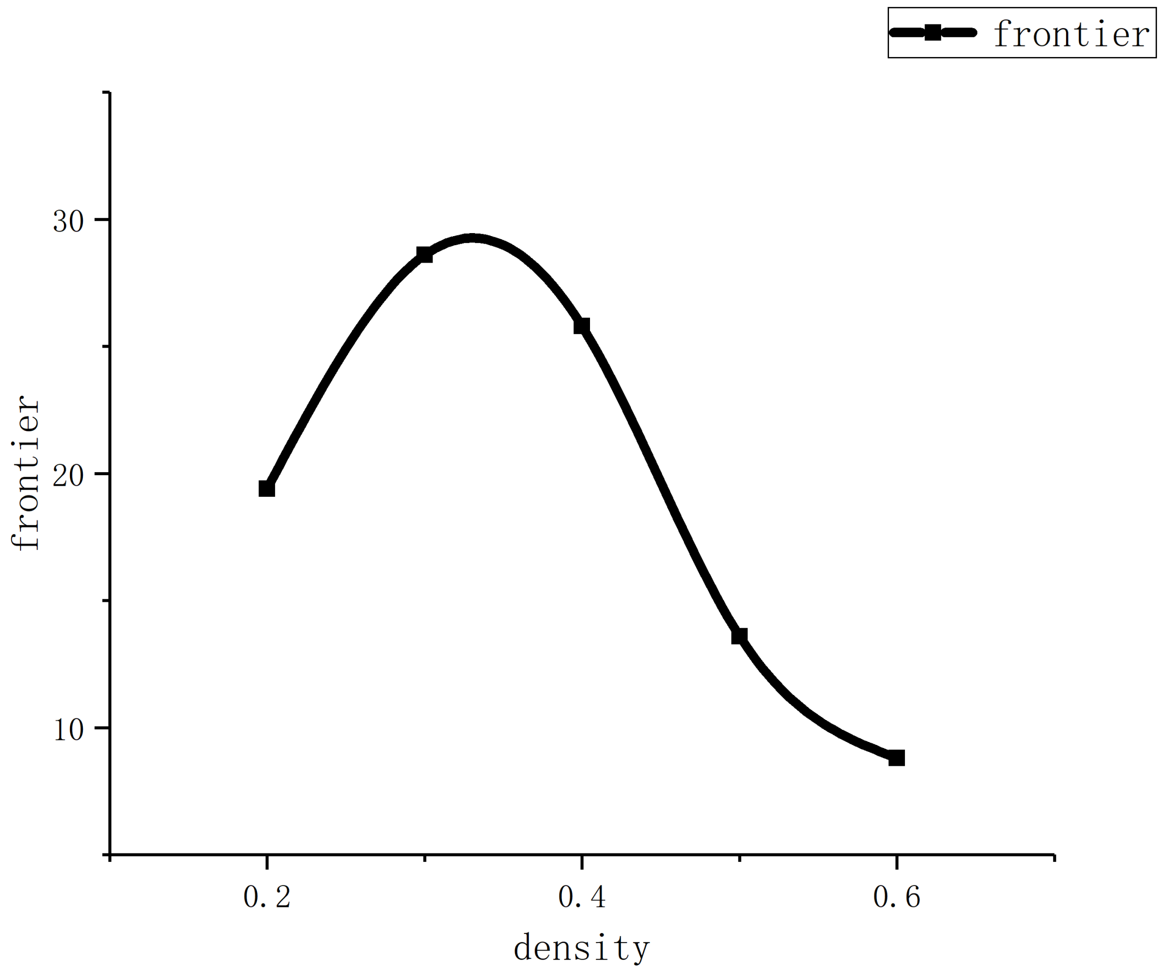


Figure 3. The length of langest chains of influences(frontier) versus the mine density. We get the maximum value at 0.35

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
            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
```

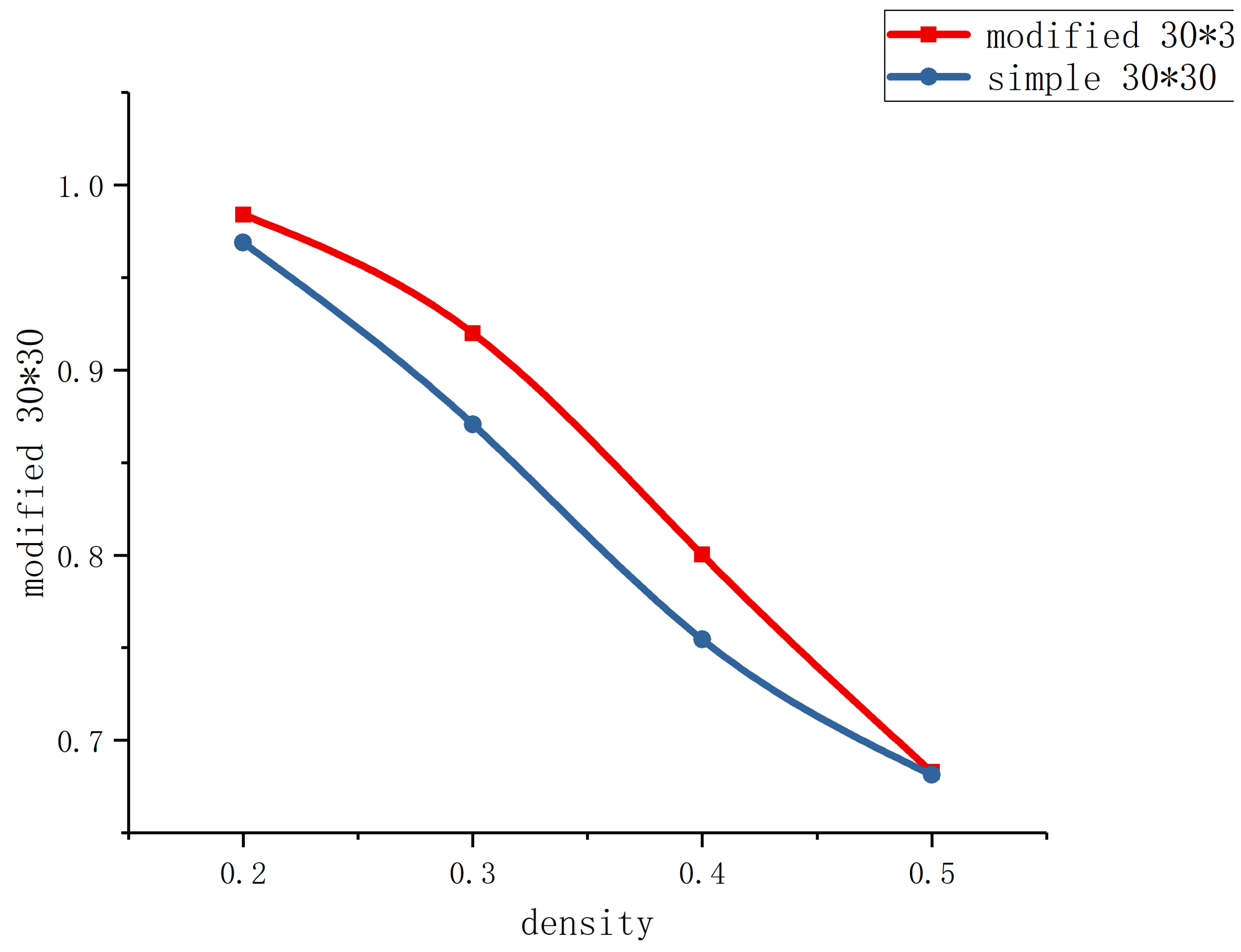


Figure 4. The comparison between the two algorithm: the blue one that is only for simple determination and guess, the red one is for the modified algorithm with both the algorithm based on pattern detemination or chains of influence combination.

- 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.

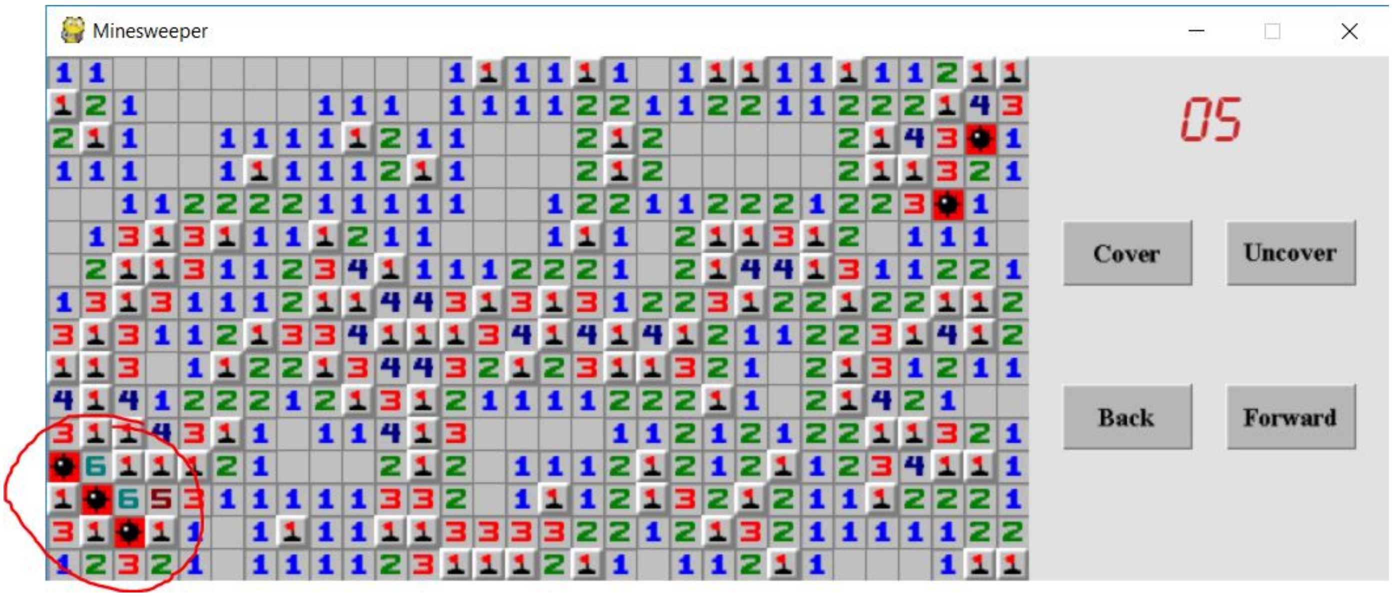


figure 5. We can see that this work not done well (this is a standard minesweeper), this is because the first step is not good, with less information, our agent made a lot of random guess, so it made a lot of errors

- 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

5 Dealing with Uncertainty

- For this question, we decided to make the changed of requirement before the agent uncovers the cells.
- When the cell contains not 0(blank) or -1(mine), it has some probability to show a symbol 'N' instead of showing the real number of it supposed to be. In this case, we need to think the cell contains the number from minimum 1 to maximum 8. However, considering the real situation, the 'N' symbol cell may have some adjacency cells that contains both 0, -1, or some number. So, depending on this situation, we give it equal probability for each scenario and based on this probability, we can find out the accurate number or just we can just pick a most likely non-mine cell to keep uncover and get more knowledge for the agent.
- In this situation, we generate a normal board and randomly choose some non-mine and non-blank cells to make them contain a smaller number than it should be. Furthermore, we want to and a filter enables the agent can have the ability to assume all the number it uncovered can contain more mines around it. If based on its knowledge base, the agent gets a highly risky cell, then choose the lowest risk probability to uncover it.
- This scenario is much similar comparison with the second one. In this situation, we only need to make the board randomly choose some none-mine and none-blank cells to add some numbers to it making it larger. Then, the following steps will be the same as the second scenario, just in a backward way.