Mine Sweeper

Chaoji Zuo(cz296)

Haotian Xu(hx105)

Xuenan Wang(xw336)

Spring 2019

Methodology

• Board Generation:

Generate a matrix of n-by-n and then randomly generate two arrays of size of m. The two arrays represent the x and y coordinate of mines location accordingly and we can label these location by *matrix[x][y] = -1*. In the game board, -3 represent out of board, which is not actual exist in normal board but as long as an out-of-board location is called, there will be a warning. -1 represent mines. 0~8 represents the number of mines in current location's neighbor. A location's neighbor is defined as its adjacent 8 blocks.

• Questions and writeup

Idea I:

1. Representation:

We assume that the game board size is ***d*d*** and the number of mines is ***m***. Data type in this game board are ***int***. Accordingly, we can draw an instance as **chart 1.1**, where ***-1*** represent a mine, ***0~8*** represent the total number of mines in adjacent blocks:

1	-1	1	0	0	1	2	2	1	0
2	2	3	1	1	1	-1	-1	2	1
1	-1	2	-1	1	1	2	2	2	-1
1	1	2	1	1	0	0	0	1	1
1	1	0	0	0	1	1	1	0	0
-1	1	0	0	0	1	-1	1	0	0
2	3	1	1	0	2	2	2	1	1
-1	3	-1	1	0	1	-1	1	1	-1
2	-1	2	2	0	1	1	1	2	2
1	2	-1	1	0	0	0	0	1	-1

Chart 1.1 Instance of a game board.

As for the knowledge base and agent action record, we use another matrix called play_ground, this play_ground is same size as game board, and is automatically generated right after the formation of game board. In this play_ground, we basically record the mines that have been flagged and possibility of exact block being a mine. An instance of

play_ground is shown as **chart 1.2**, where **-2** represents the blocks that haven't been explored, ***0*** represents that the possibility of exact this block being a mine is zero, and of course, ***0.143*** represents that the possibility of exact this block being a mine is 0.125(=1/8).

-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
0.143	0.143	0.143	-2	-2	-2	-2	-2	-2	-2
0.143	0	0.143	-2	-2	-2	-2	-2	-2	-2
0.143	0.143	0	0	0	-2	-2	-2	-2	-2
-2	-2	0	0	0	-2	-2	-2	-2	-2
-2	-2	-2	-2	0	-2	-2	-2	-2	-2
-2	-2	-2	-2	0	-2	-2	-2	-2	-2
-2	-2	-2	-2	0	-2	-2	-2	-2	-2
-2	-2	-2	-2	0	-2	-2	-2	-2	-2

Chart 1.2 Instance of a play_ground.

In **chart 1.2**, assume we define point (0, 0) is the top-left block and we explore (5, 3) first. Because the value of block (5, 3) in game board is 0, we call a function to examine the if there are also block are zero in the adjacent blocks. If there are also some blocks that value is zero, we explore them as well. After that, we will explore the adjacent block on the up-left direction of (5, 3). Since the value of this block in the game board is 1, which indicates that there are one mine in its adjacent blocks. Considering we already know that (5, 3) is not a mine, so the possibility of each other seven unknown block being a mine is 0.143(=1/7).

The whole purpose of designing this play_ground is that we can get the possibility of blocks being a mine and accordingly, we can choose those blocks with low possibilities to explore, which means that these possibilities is our knowledge base and they will update right after every move we make.

2. Inference:

Every time a block is explored, the agent will also calculate the possibilities of adjacent blocks being mines and assign those value to corresponding blocks in play_ground. If there is already a value, then we need to calculate the possibility of these two event at least one will happen. So basically, it's [1 - (1 - old_possibility)*(1 - new_possibility)]. After these calculations and value assignments, we will sort all of the possibility in play_ground, and the least one(except zero) will be the next block to be explored.

3. Decisions:

The agent will choose the least possibility in play_ground to explore and after exploring it, if not a mine, this block will be assigned zero(it makes sense because after explore, this block can not be a mine). Then it will repeat calculate adjacent possibilities and sort possibilities so that the knowledge base can be updated and we can know which block is the next target.

Idea II:

1. Representation:

Set the dimension for the mine sweeper, which will be a N x N square, and the number of mines will be given to the map. Randomly distributing those mines, which would be -1 in the map. Then we would find out the number of mines around an empty spot and set a number to it. There is no fancy thing for building the map.

2. Inference:

Idea II take this mine sweeper problem as a CSP. For example, take the trivial map — a 3 x 3 map with only one mine in it.

A	B	C
D	3	E
F	G	H

In this case we have 3 in the center, and 8 unknown spot around it. If we take a boolean expression for a spot, then each point can either be 1 or 0, where 1 represents there is a mine and 0 represents there is not.

Therefore, we would have a numerical expression for A + B + C + D + E + F + G + H = 3, or there will only three two value expression that add up to 1, for example A + E = 1, B + C = 1, and D + H = 1, which indicates the rest would add to 0.

When we have more and more grids like this one, we would have more information to set possible 0/1 to those empty spots.

If we take mine sweeper as a CSP, then the number show on the non-mine-spot will the constraint for it surrounding. Then there will be 3 typical cases:

A. The number shown is 0, which means that all the spots around the central place will be safe.

A	B	C
D	0	E
F	G	H

In this figure, the surrounding places will be the safe.

B. The number shown is equal tot the empty spot (which means there is no number or -1 shown on that point), then this will indicate all the empty spots around the central place has a mine in it.

1	A	2
2	3	C
B	2	1

In this example, A, B, and C will definitely have mines.

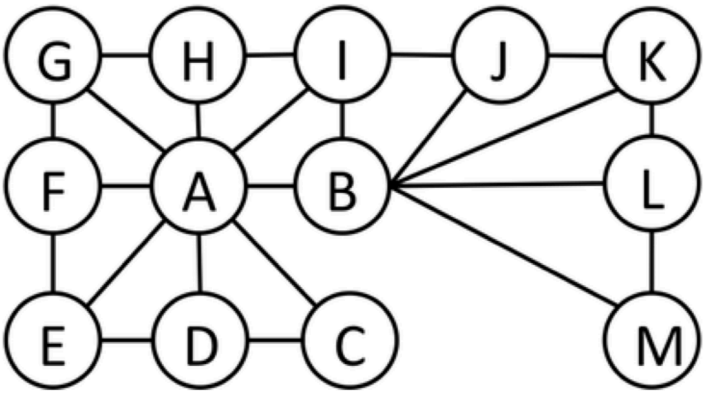
C. The number, *M*, shown in the spot is less than the *N* empty spots around the central place. In these case we will have N choose M possible options for us to set 0/1

For cases A and B, they are kind of the base case in this method. We mainly focus on case C.

When we have a sufficient amount information, we could use backtracking and forward-checking to try to solve the problem.

To better perform this notion, I choose graph as the data structure. Each point the agent choose will generate a complete connected component. Components may share vertices between each other.

The graph may look like this, where each letter represents an empty spot and the edges are the constraints.



3. Decisions:

First: forward-checking: As long as we encounter case A or case B, we would make some spot in a well-defined state: it is either safe (there is no mine) or dangerous (there is a mine). We keep those as kind of complementary information, which could tell us where could definitely be a safe place or a dangerous place. Based on them we can cut off the possible distribution of 0/1.

Second: Backtracking: when we have a certain amount of information, there will be some spots sharing neighbors. However, those spot might have different number shown on the map. Therefore, the way we assign 0/1 will have some constraints. we can take either one as starting place (Choosing the one with biggest number would give a better result). Then we assign one number at a time; meanwhile, we have to keep the consistency for all the other place. If the current assigned node with value 1 contradicts to one the component consistency, the we will keep that node with value 0.

Component consistency: Each component will have an associated number representing the number of mines in the surrounding. No matter what kind of distribution of mines around the center, the total number that mines add up should be equal to the number shown on the map.

Pseudocode for Component Consistency and the bracktracking:

```
function Component_Consistency( component_node ):
        return associated_number(component_node) == mines(component_node)
function backTracking( graph ):
        for component in graph:
```

for vertex in component:

value(vertex) = 1 if Component_Consistency(other_component)

else: value(vertex) = 0

if Component_Consistency(component): break

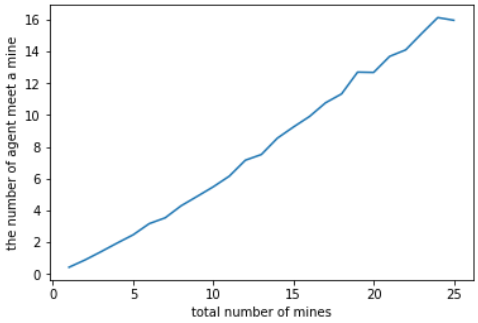
4. Efficiency: the worst case is that we have to run all the spot in a n x n map. Then for there will be n^2 component node in the graph, each one will have maximal 8 adjacent vertices. The total time will be nΣ(k*8) = 8 * (n^2 + n) / 2 = 4 * (n^2 + n).

This algorithm does not perfectly solve the mineSweeper problem. Because backtracking requires a certain amount information, which we do not have at the first place. Moreover, if the mines are collectively distributed in one corner of the map, and my agent randomly pick place at other place in the very beginning. The agent will face a great chance of jumping into a mine spot in the middle of process. Because backtracking is a brute-force algorithm, it is not guarantee for generate the optimal result, which means there is a situation that there are multiple ways to satisfy the constraint, and my agent choose the wrong one.

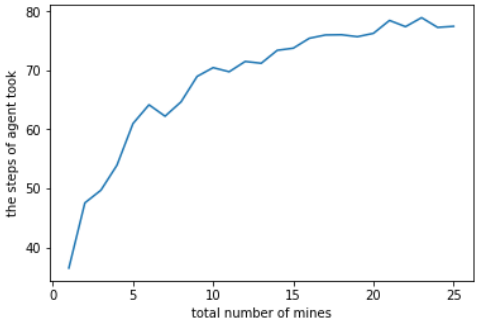
5. Performances:

First, let the size of map be fixed, then the number of agent encountering a mine will increase as the total number of mine increase.

Based on the image, there is basic a linear relationship between the # mines and the chance of getting into a mine spot.

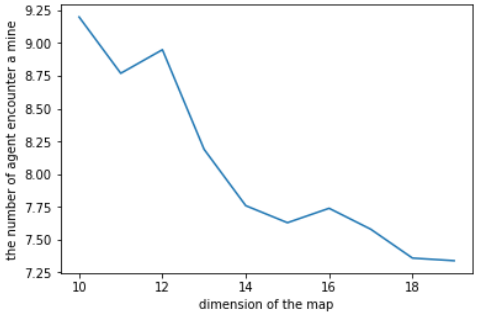


Second, the step that the agent takes. As the number of mines increases, the steps will converge to a constant, based on the image, the constant would be 80.

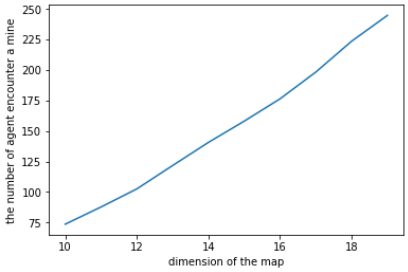


Third, Make the #mines fixed, changing the size of map. #mines = 15.

As the size becomes larger, the chance that the agent will meet a mine will go down.



Fourth. As the size increase, the steps for agent to solve the maze also increment.



6. Improvements:

Use a strategy to play the game. Instead of randomly choosing a place at very beginning, choose those places that will the most information about the map. For example, for a 5x5 map, we can choose the 4 corner spots, and these 4 spots will bring the whole information of the entire map, even though the information may not help us to shrink the possibility.

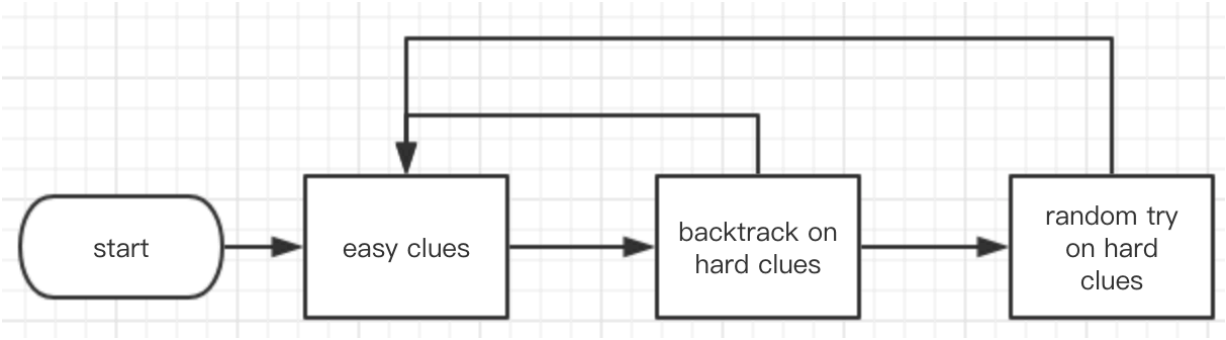
Idea III:

1. Representation:

Like the other ideas, I also used matrix. And -1 represent mine, -2 represent unknown block.

2. Inference:

When I collect a new clue, I would store it in a heap, and every time I find a clue I cannot get new information from it, I would store it in another heap name “cannot solve blocks”. After I processing all the easy blocks, I would try to do backtracking on these hard blocks. If I still cannot get a great result, I would random try to solve it. What’s more, every time my agent find a new mine or clue, it would record it and update the knowledge base of its nearby. The whole flow can be showed as figure III.1.



III.1 flow chart of idea III's agent

3. Decisions:

In my program, I would always try to figure out the clue whose value is small, because I have the heap structure. And every time I figure out a new clue, I would add it's nearby blocks to the heap. There still exists the cannot solvable situation. So my agent has to take the risks.

4. Performance:

In most case, I agree with my agent, because most times it's right. And it would surprise me because it calculate mach faster than me, it's hard for me to do back tracking by my brain. But still, I think my backtracking algorithm is not perfect, because every time it would list all the possible situation. I think a perfect algorithm can return a value each time it find a certainly conclusion, in this way, the useless calculation can reduce a lot.

I also compare the performance of backtracking, I found that it really can reduce some unwanted error. It average can reduce the times of touching a mine 2 to 3 times, which is really remarkable for a minesweeper game.

5. Performance:

The Figure III.2 shows the relationship between number of mines and final score(safely identified mines / total mines). I thinks this result is pretty great, at least %90 mines can be found. And as you can see, when the number of mines increase, the score would decrease.

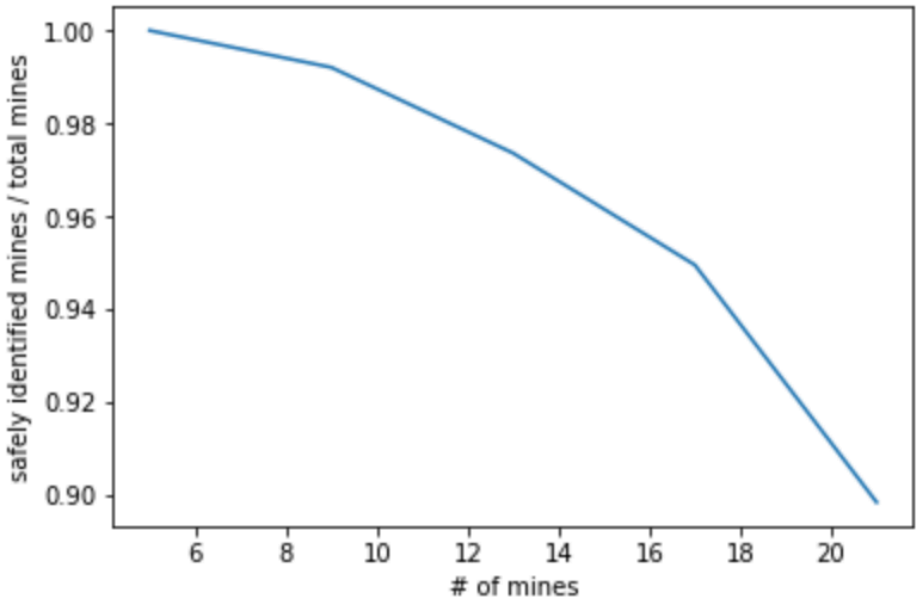


Figure III.2 score vs number of mines (board size is 10)

I also try to fixed the relative rate of mines and see what's the change of score, and the result is showed in Figure III.3. As you can see, the score would increase a little when the size increase. So we conclude that the density of mines would be the most relative factor for hard minesweeper.

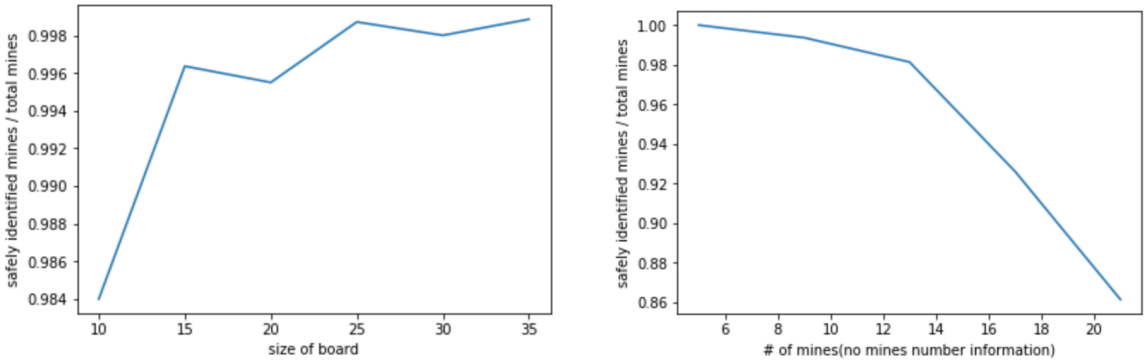


Figure III.3 score vs size of board (density of mines is 0.1)

6. Efficiency:

As I mentioned in question 5. The backtracking algorithm is not perfect so I add some constraints to reduct the running time. It is undoubtedly an implement program. I have some ideas to improve it, for example, produces the connected graph separately, break the recursion once find the certain value. But these ideas are not easy to achieve.

7. Improvement:

The performance would increase a little if telling the agent the number of mines in advance. And I think it is a necessary condition. Because the real minesweeper game would also tell you the number of mines. In this way, you can know which hypothesis is right in some special case. What's more, the time would be reduce if the agent have mines' information cause it reduce some process. Figure III.4 shows the result.

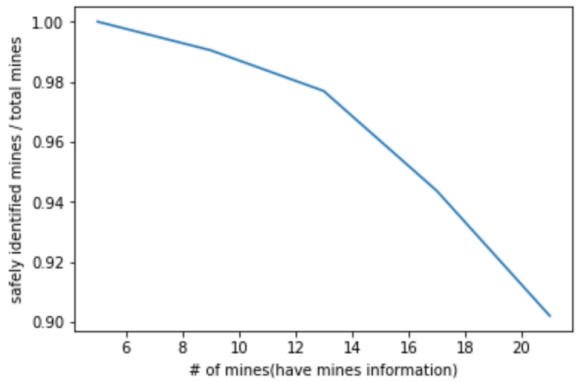


Figure III.4 compare of weather knowing the mines number

- Chains of Influence

Every time the agent open a new mine, its neighbors' knowledge base would update, and also it may bring some new clue, then the agent would search its neighbor, try to find whether there are more blocks can be solved. This strategy can make sure the agent gain more information fast.

The start point would influence the length of the longest chain. A certain board, if you start from different point, the result would be different too.

Longer of the chain. Better of the efficiency. So in our agent, we try best to make the chain longer. Every time the knowledge base change, the length of node list would increase.

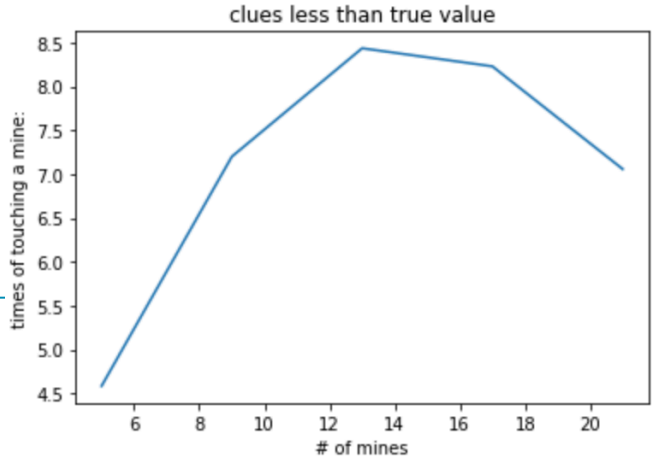
Yes, we can find a board that yields particularly long chains of influence. Once the mines cut off the road of clues. In other words, more mines, shorter chains.

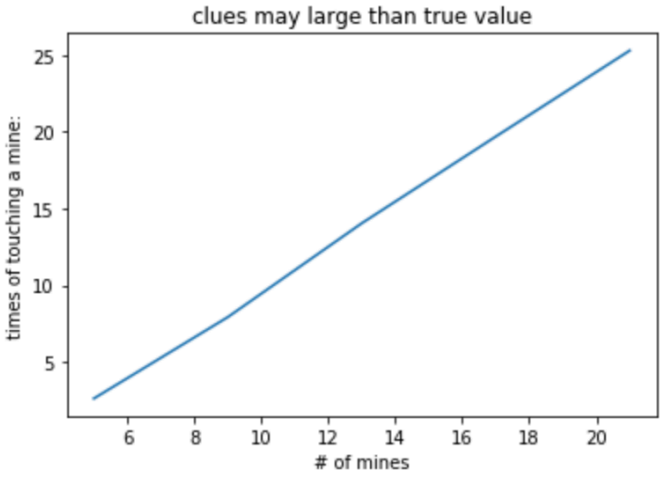
Theoretically, the longest chain can travel the whole board, because it can keep updating the knowledge base until get the goal state.

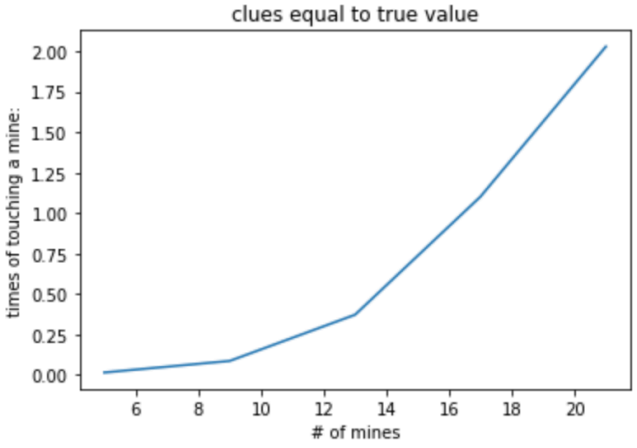
If an agent can always find the longest way when the game begin, it can always solve the board more efficiently. Actually, we have a condition of our agent is that make sure the first step would always be zero, which means it can at least open its nearby blocks and get more information.

It is no very hard to solve minesweeper, but it would be harder if you want to consider all the possible situation and find the most efficiency way.

- Dealing with Uncertainty







I try the second and third uncertain case and get a surprise result. I use the times of touching a mine to represent the score of different model. And if the clues may larger than the true value, the performance would be really bad, the agent even would touch a mine

more than one times. This is because my backtracking algorithm would need the clue. And in the less case, the agent would be more likely to touch a mine, because it this situation, there would be more 0 in the board than real. So the agent would thought there may be no mines nearby. Unfortunately it would touch a mine.

- Contribution:
  - Chaoji Zuo: Regular Question + Bonus Question
  - Haotian Xu: Regular Question
  - Xuenan Wang: Regular Question

- Appendix: Source Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt
import random
import sys
sys.setrecursionlimit(1000000)

class Map:
  def __init__(self, d:'dimention of Map', n:'number of mines'):
        self.dimention = d
        self.mine_number = n
        self.map = np.zeros((self.dimention, self.dimention))

  def generate_map(self):
        v_mine = np.array([])
        h_mine = np.array([])
        print('self.dimention:', self.dimention)
        print('self.mine_number:', self.mine_number)
        print('map:\n', self.map)
        for i in range(self.mine_number):
```

```
                # print('Here is for-loop!')
                v_mine = np.append(v_mine, random.randint(0, self.dimention-1))
                h_mine = np.append(h_mine, random.randint(0, self.dimention-1))
        print('v_mine:', v_mine)
        print('h_mine:', h_mine)
        for i in range(0, self.mine_number):
                self.set_mine(int(v_mine[i]), int(h_mine[i]))
                # still not right, this should be return to is_mine and go forward
whenever is satisfied.
        print('mine map:\n', self.map)

    def is_mine(self, x:'x coordinate', y:'y coordinate'):
        if self.map[x][y] == -1:
                return True
        else:
                return False

    def set_mine(self, x:'x coordinate', y:'y coordinate'):
        if self.is_mine(x, y) == False:
                self.map[x][y] = -1
        elif self.is_mine(x, y) == True:
                random_x = random.randint(0, self.dimention-1)
                random_y = random.randint(0, self.dimention-1)
                return self.set_mine(random_x, random_y)
        else:
                print('There is an error in return of is_mine function!')

    def go_east(self, x:'x coordinate', y:'y coordinate'):
        return x+1, y

    def go_west(self, x:'x coordinate', y:'y coordinate'):
        return x-1, y

    def go_north(self, x:'x coordinate', y:'y coordinate'):
```

```
            return x, y-1

        def go_south(self, x:'x coordinate', y:'y coordinate'):
            return x, y+1

        def get_neighbor(self, x:'x coordinate', y:'y coordinate') -> 'return a dictionary in which
contains map relation of direction and box value':
        # Agent should NOT have direct access to this function
            """
            sample map
            @  -3 **out of map**
            @  -2 **signal for mine**
            @  -1 **mine**
            @   0 **no mine or neighbor no mine**
            @ 1-8 **number of neignbor mines**

              x0 x1 x2 x3 x4 outside
             y0 1 -1 -1  2  1 -3
             y1 2  3  4 -1  1 -3
             y2 1 -1  3  3  2 -3
             y3 1  2 -1  2 -1 -3
             y4 0  1  1  2  1 -3
              -3 -3 -3 -3 -3 -3
            """
            neighbor = {'north' : -3, 'north_east' : -3, 'east' : -3, 'south_east' : -3, 'south' : -3,
'south_west' : -3, 'west' : -3, 'north_west' : -3}
            if x-1>=0 and x+1<=self.dimention-1 and y-1>=0 and y+1<=self.dimention-1:
                # x1-x3 and y1-y3
                neighbor['north'] = self.map[x][y-1]
                neighbor['north_east'] = self.map[x+1][y-1]
                neighbor['east'] = self.map[x+1][y]
                neighbor['south_east'] = self.map[x+1][y+1]
                neighbor['south'] = self.map[x][y+1]
                neighbor['south_west'] = self.map[x-1][y+1]
```

```
            neighbor['west'] = self.map[x-1][y]
            neighbor['north_west'] = self.map[x-1][y-1]
        elif x-1>=0 and x+1<=self.dimention-1 and y+1<=self.dimention-1:
            # y0 aside (x0,y0) and (x4,y0) [north side]
            neighbor['north'] = -3
            neighbor['north_east'] = -3
            neighbor['east'] = self.map[x+1][y]
            neighbor['south_east'] = self.map[x+1][y+1]
            neighbor['south'] = self.map[x][y+1]
            neighbor['south_west'] = self.map[x-1][y+1]
            neighbor['west'] = self.map[x-1][y]
            neighbor['north_west'] = -3
        elif x-1>=0 and y-1>=0 and y+1<=self.dimention-1:
            # x4 aside (x4,y0) and (x4,y4) [east side]
            neighbor['north'] = self.map[x][y-1]
            neighbor['north_east'] = -3
            neighbor['east'] = -3
            neighbor['south_east'] = -3
            neighbor['south'] = self.map[x][y+1]
            neighbor['south_west'] = self.map[x-1][y+1]
            neighbor['west'] = self.map[x-1][y]
            neighbor['north_west'] = self.map[x-1][y-1]
        elif x-1>=0 and x+1<=self.dimention-1 and y-1>=0:
            # y4 aside (x0,y4) and (x4,y4) [south side]
            neighbor['north'] = self.map[x][y-1]
            neighbor['north_east'] = self.map[x+1][y-1]
            neighbor['east'] = self.map[x+1][y]
            neighbor['south_east'] = -3
            neighbor['south'] = -3
            neighbor['south_west'] = -3
            neighbor['west'] = self.map[x-1][y]
            neighbor['north_west'] = self.map[x-1][y-1]
        elif x+1<=self.dimention-1 and y-1>=0 and y+1<=self.dimention-1:
            # x0 aside (x0,y0) and (x0,y4) [west side]
```

```
            neighbor['north'] = self.map[x][y-1]
            neighbor['north_east'] = self.map[x+1][y-1]
            neighbor['east'] = self.map[x+1][y]
            neighbor['south_east'] = self.map[x+1][y+1]
            neighbor['south'] = self.map[x][y+1]
            neighbor['south_west'] = -3
            neighbor['west'] = -3
            neighbor['north_west'] = -3
        elif x == 0 and y == 0:
            # (x0,y0)
            neighbor['north'] = -3
            neighbor['north_east'] = -3
            neighbor['east'] = self.map[x+1][y]
            neighbor['south_east'] = self.map[x+1][y+1]
            neighbor['south'] = self.map[x][y+1]
            neighbor['south_west'] = -3
            neighbor['west'] = -3
            neighbor['north_west'] = -3
        elif x == self.dimention-1 and y == 0:
            # (x4,y0)
            neighbor['north'] = -3
            neighbor['north_east'] = -3
            neighbor['east'] = -3
            neighbor['south_east'] = -3
            neighbor['south'] = self.map[x][y+1]
            neighbor['south_west'] = self.map[x-1][y+1]
            neighbor['west'] = self.map[x-1][y]
            neighbor['north_west'] = -3
        elif x == self.dimention-1 and y == self.dimention-1:
            # (x4,y4)
            neighbor['north'] = self.map[x][y-1]
            neighbor['north_east'] = -3
            neighbor['east'] = -3
            neighbor['south_east'] = -3
```

```
                neighbor['south'] = -3
                neighbor['south_west'] = -3
                neighbor['west'] = self.map[x-1][y]
                neighbor['north_west'] = self.map[x-1][y-1]
            elif x == 0 and y == self.dimention-1:
                # (x0,y4)
                neighbor['north'] = self.map[x][y-1]
                neighbor['north_east'] = self.map[x+1][y-1]
                neighbor['east'] = self.map[x+1][y]
                neighbor['south_east'] = -3
                neighbor['south'] = -3
                neighbor['south_west'] = -3
                neighbor['west'] = -3
                neighbor['north_west'] = -3
            else:
                print('There is an error in Map.get_neighbor function!')
            return neighbor

        def get_box_value(self, x:'x coordinate', y:'y coordinate') -> 'return an integer which is the mine number of neighborhood':
            neighbor = self.get_neighbor(x, y)
            box_mine_number = 0
            for direction in neighbor:
                if neighbor[direction] == -1:
                    box_mine_number += 1
            return box_mine_number

        def is_mine(self, x:'x coordinate', y:'y coordinate'):
            if self.map[x][y] == -1:
                return True
            elif self.map[x][y] >= 0:
                return False
            else:
                print('This is an illegal access!')
```

```
    def set_ready(self):
        self.generate_map()
        for y in range(0, self.dimention):
            for x in range(0, self.dimention):
                if self.is_mine(x, y) == False:
                    self.map[x][y] = self.get_box_value(x, y)
                elif self.is_mine(x, y) == True:
                    self.map[x][y] = -1
                else:
                    print('Problem in function of set_value!')


class PlayGround():
    def __init__(self, d:'dimention of Map', n:'number of mines', t:'threshold for mines'):
        # dimention of PlayGround must keep same as Map
        self.dimention = d
        self.mine_number = n
        self.play_ground = np.zeros((self.dimention, self.dimention))
        self.play_ground.fill(-2)
        self.mine_map = Map(self.dimention, self.mine_number)
        self.mine_map.set_ready()
        self.next_step = [-1, -1]
        self.wrong_step = 0
        self.mine_threshold = t
        print('play_ground:\n', self.play_ground)
        """
        sample play_ground
        @  -3 **out of map**
        @  -2 **origin state**
        @  -1 **flag a mine**
        @ 0-8 **number in the same coordinate in mine_map**

          x0 x1 x2 x3 x4 outside
         y0 1 -1 -1  2  1 -3
```

```
        y1 2  3  4 -1  1 -3
        y2 1 -1  3  3  2 -3
        y3 1  2 -1  2 -1 -3
        y4 0  1  1  2  1 -3
          -3 -3 -3 -3 -3 -3
        """

    def make_guess(self) -> 'return random coordinate x, y':
        x = random.randint(0, self.dimention-1)
        y = random.randint(0, self.dimention-1)
        print('guess coordinate is: (', x, ', ', y, ')')
        if self.is_visited(x, y) == True:
            return self.make_guess()
        else:
            return [x, y]

    def expand_box(self, x:'x coordinate', y:'y coordinate'):
        '''
        self.play_ground[x-1][y-1] = self.mine_map[x-1][y-1]
            self.play_ground[x][y-1] = self.mine_map[x][y-1]
            self.play_ground[x+1][y-1] = self.mine_map[x+1][y-1]
            self.play_ground[x-1][y] = self.mine_map[x-1][y]
            self.play_ground[x+1][y] = self.mine_map[x+1][y]
            self.play_ground[x-1][y+1] = self.mine_map[x-1][y+1]
            self.play_ground[x][y+1] = self.mine_map[x][y+1]
            self.play_ground[x+1][y+1] = self.mine_map[x+1][y+1]
            '''
        if self.is_zero(x-1, y-1) == True:
            self.expand_box(x-1, y-1)
        elif self.is_zero(x, y-1) == True:
            self.expand_box(x, y-1)
        elif self.is_zero(x+1, y-1) == True:
            self.expand_box(x+1, y-1)
        elif self.is_zero(x-1, y) == True:
```

```
                self.expand_box(x-1, y)
        elif self.is_zero(x+1, y) == True:
                self.expand_box(x+1, y)
        elif self.is_zero(x-1, y+1) == True:
                self.expand_box(x-1, y+1)
        elif self.is_zero(x, y+1) == True:
                self.expand_box(x, y+1)
        elif self.is_zero(x+1, y+1) == True:
                self.expand_box(x+1, y+1)

'''
def is_expandable(self, x:'x coordinate', y:'y coordinate'):
        if self.mine_map[x][y] == 0:
                return True
        else:
                return False
'''

def is_visited(self, x:'x coordinate', y:'y coordinate'):
        if x < 0 or x >= self.dimention:
                return False
        if y < 0 or y >= self.dimention:
                return False
        if self.play_ground[x][y] >= -1 and self.play_ground[x][y] <= 0:
                return True
        else:
                return False

def is_mine(self, x:'x coordinate', y:'y coordinate'):
        if x < 0 or x >= self.dimention:
                return False
        if y < 0 or y >= self.dimention:
                return False
        if self.mine_map.map[x][y] == -1:
```

```
                return True
        else:
                return False

def is_zero(self, x:'x coordinate', y:'y coordinate'):
        if x < 0 or x >= self.dimention:
                return False
        if y < 0 or y >= self.dimention:
                return False
        if self.mine_map.map[x][y] == 0:
                self.play_ground[x][y] = 0
                return True
        else:
                return False

def count_not_visited_neighbor(self, x:'x coordinate', y:'y coordinate'):
        i = 0
        # list_available_neighbor = np.array([[],[]])
        available_neighbor_x = np.array([])
        available_neighbor_y = np.array([])
        '''
        if self.is_visited(x-1, y-1) == True:
                i += 1
                list_available_neighbor = np.append(list_available_neighbor[:][0], x-1)
                list_available_neighbor = np.append(list_available_neighbor[:][1], y-1)
        if self.is_visited(x, y-1) == True:
                i += 1
                list_available_neighbor = np.append(list_available_neighbor[:][0], x)
                list_available_neighbor = np.append(list_available_neighbor[:][1], y-1)
        if self.is_visited(x+1, y-1) == True:
                i += 1
                list_available_neighbor = np.append(list_available_neighbor[:][0], x+1)
                list_available_neighbor = np.append(list_available_neighbor[:][1], y-1)
        if self.is_visited(x-1, y) == True:
```

```
            i += 1
            list_available_neighbor = np.append(list_available_neighbor[:][0], x-1)
            list_available_neighbor = np.append(list_available_neighbor[:][1], y)
    if self.is_visited(x+1, y) == True:
            i += 1
            list_available_neighbor = np.append(list_available_neighbor[:][0], x+1)
            list_available_neighbor = np.append(list_available_neighbor[:][1], y)
    if self.is_visited(x-1, y+1) == True:
            i += 1
            list_available_neighbor = np.append(list_available_neighbor[:][0], x-1)
            list_available_neighbor = np.append(list_available_neighbor[:][1], y+1)
    if self.is_visited(x, y+1) == True:
            i += 1
            list_available_neighbor = np.append(list_available_neighbor[:][0], x)
            list_available_neighbor = np.append(list_available_neighbor[:][1], y+1)
    if self.is_visited(x+1, y+1) == True:
            i += 1
            list_available_neighbor = np.append(list_available_neighbor[:][0], x+1)
            list_available_neighbor = np.append(list_available_neighbor[:][1], y+1)
    return i, list_available_neighbor
    '''
    if self.is_visited(x-1, y-1) == True:
            i += 1
            available_neighbor_x = np.append(available_neighbor_x, x-1)
            available_neighbor_y = np.append(available_neighbor_y, y-1)
    if self.is_visited(x, y-1) == True:
            i += 1
            available_neighbor_x = np.append(available_neighbor_x, x)
            available_neighbor_y = np.append(available_neighbor_y, y-1)
    if self.is_visited(x+1, y-1) == True:
            i += 1
            available_neighbor_x = np.append(available_neighbor_x, x+1)
            available_neighbor_y = np.append(available_neighbor_y, y-1)
    if self.is_visited(x-1, y) == True:
```

```
                i += 1
                available_neighbor = np.append(available_neighbor_x, x-1)
                available_neighbor = np.append(available_neighbor_y, y)
        if self.is_visited(x+1, y) == True:
                i += 1
                available_neighbor = np.append(available_neighbor_x, x+1)
                available_neighbor = np.append(available_neighbor_y, y)
        if self.is_visited(x-1, y+1) == True:
                i += 1
                available_neighbor = np.append(available_neighbor_x, x-1)
                available_neighbor = np.append(available_neighbor_y, y+1)
        if self.is_visited(x, y+1) == True:
                i += 1
                available_neighbor = np.append(available_neighbor_x, x)
                available_neighbor = np.append(available_neighbor_y, y+1)
        if self.is_visited(x+1, y+1) == True:
                i += 1
                available_neighbor = np.append(available_neighbor_x, x+1)
                available_neighbor = np.append(available_neighbor_y, y+1)
        return i, available_neighbor_x, available_neighbor_y

    '''
    def make_move(self, x:'x coordinate', y:'y coordinate'):
        if self.is_mine(x, y) == True:
            print('You LOSE!! Try again!!')
        elif self.is_zero(x, y) == True:
            self.expand_box(x, y)
        else:
    '''

    # conditional probability - P(p_new | p_pre)
    def calculate_possibility(self, p_new:'possibility of new event', p_pre:'possibility of
origin'):
        # return ((p_new+p_pre)-p_new*p_pre)/p_pre
```

```
            return 1-((1-p_new)*(1-p_pre))

        def label_possibility(self, x:'x coordinate', y:'y coordinate'):
            if self.is_zero(x, y) == False:
                if self.is_mine(x, y) == False:
                    # this bos is visited and testified that is not a mine, therefore the
possibility of this box being a mine is 0
                    self.play_ground[x][y] = 0
                    visited, available_neighbor_x, available_neighbor_y =
self.count_not_visited_neighbor(x, y)
                    for neighbor_x in range(0, len(available_neighbor_x)):
                        for neighbor_y in range(0, len(available_neighbor_y)):
                            if self.is_visited(neighbor_x, neighbor_y) == False:
                                self.play_ground[neighbor_x][neighbor_y] =
self.mine_map.map[x][y]/visited
                            else:
                                self.play_ground[neighbor_x][neighbor_y] =
self.calculate_possibility(self.mine_map.map[x][y]/visited, self.play_ground[neighbor_x]
[neighbor_y])


                    '''
                    # below is wrong
                    if self.is_visited(x, y) == True:
                        self.play_ground[x][y] =
self.calculate_possibility(self.mine_map[x][y]/(8-visited), self.play_ground[x][y])
                    else:
                        self.play_ground[x][y] = self.mine_map[x][y]/(8-visited)
                    '''

        def label_mine(self, x:'x coordinate', y:'y coordinate'):
            x = int(x)
            y = int(y)
            self.play_ground[x][y] = -1
```

```
def sort_possibility(self):
        least = np.zeros(3)
        largest = np.zeros(3)
        for y in range(0, self.dimention):
                for x in range(0, self.dimention):
                        if self.play_ground[x][y] < least[0]:
                                least[0] = self.play_ground[x][y]
                                least[1] = x
                                least[2] = y
                        elif self.play_ground[x][y] > largest[0]:
                                largest[0] = self.play_ground[x][y]
                                largest[1] = x
                                largest[2] = y
        return least, largest

def reach_mine_threshold(self, largest):
        if largest[0] >= self.mine_threshold:
                self.label_mine(largest[1], largest[2])
                return True
        else:
                return False

def is_game_end(self):
        i = 0
        for y in range(0, self.dimention):
                for x in range(0, self.dimention):
                        if self.play_ground[x][y] == 0 or self.play_ground[x][y] == -1:
                                i += 1
        if i == self.dimention * self.dimention:
                return True
        else:
                return False
```

```
    def run(self, x:'x coordinate', y:'y coordinate'):
        x = int(x)
        y = int(y)
        print('this time coordinate: (', x, ', ', y, ')')
        print('now play_ground is:\n', self.play_ground)
        if self.is_game_end() == True:
            print('mine_map:\n', self.mine_map.map)
            print('play_ground:\n', self.play_ground)
        else:
            if self.is_mine(x, y) == True:
                self.label_mine(x, y)
                new_x, new_y = self.make_guess()
                return self.run(new_x, new_y)
            elif self.is_zero(x, y) == True:
                self.expand_box(x, y)
                least, largest = self.sort_possibility()
                if self.reach_mine_threshold(largest) == True:
                    self.label_mine(largest[1], largest[2])
                return self.run(least[1], least[2])
            else:
                self.label_possibility(x, y)
                least, largest = self.sort_possibility()
                if self.reach_mine_threshold(largest) == True:
                    self.label_mine(largest[1], largest[2])
                return self.run(least[1], least[2])


if __name__ == '__main__':
    '''
    my_map = Map(10, 15)
    my_map.set_ready()
    print('My Map:\n', my_map.map)
    '''
    my_board = PlayGround(5, 5, 0.8)
    x, y = my_board.make_guess()
```

```
my_board.run(x, y)
```