Mine Sweeper

Bharath Reddy

1.Representation:

1.1 Game Modeling:

1.1.1 Basic Game Knowledge:

1. The game consists of a square box further divided into smaller boxes.

2. Each of the smaller boxes can be termed as a field.

3. The field can exist in three of the following states:

   1. Closed
   2. Open
   3. Flaged

4. A closed field can be opened or flaged.

5. If a field with a mine is opened then the game ends with you losing.

6. If you manage to avoid all the field with mines and open all the ones with out any mine then you win.

1.1.2 Approach:

In our version of implementing the game firstly, we created a template for each field as shown in the code snippet below:

```
#every element of the grid template:
class gridElement:
    def __init__(self):
        self.location = ()
        self.bomb = False
        self.weight = 0
        self.flag = False
        self.open = False
```

Each field has 5 properties each denoting a specific property of the field.

Properties and functions:

Location- helps us identify the field uniquely in the gird.

Bomb- tells us if there is a bomb in the field or not.

Flag- tells us if a field had been flaged or not.

Open- tells us if a field has been opened or not.

Weight- gives us the number of adj field with mines.

1.1.3 Game Play

We first start of by creating a new field with a specific number of bombs.

```
353 #makes a initial template of the grid
354 def makeGrid(dim):
355     grid = [[gridElement() for j in range(dim)] for i in range(dim)]
356     for i in range (dim):
357         for j in range (dim):
358             grid[i][j].location = (i,j)
359     return grid
360
361 #sets bombs inside the grid
362 def setBombs(grid,dim):
363
364     count = floor((dim*dim)/10)
365
366     while count > 0:
367         i = randint(0,dim-1)
368         j = randint(0,dim-1)
369
370         if(grid[i][j].bomb == False):
371             grid[i][j].bomb = True
372             count-=1
373     return grid
```

Our initial (7x7) grid will look something like this:

[0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 1, 1, 1, 0]

[1, 1, 1, 1, 69, 1, 0]

[2, 69, 1, 1, 1, 1, 0]

[69, 2, 1, 0, 0, 0, 0]

[1, 1, 0, 0, 0, 1, 1]

[0, 0, 0, 0, 0, 1, 69]

As seen in line 354 we call a makeGrid() function that creates a gird of dimension “dim” with multiple field. We then call the setBombs() function which sets a certain numbers of fields with bombs. The number of bombs is denoted by the code in like 364. We then place the bombs in random places with in the grid, this is achieved by setting the bomb property of the field to “69”. We then store this grid in a “referenceGrid” variable. We make sure that this is not available to the knowledge base by separating the referenceGrid and the updatedGrid.

Now after the game has been set up we now create a new grid that starts of default with no bombs in the location. As we would in a real world scenario when no information is presented to us.

```
51  def mazeSolver(dim):
52
53      global updatedGrid
54      updatedGrid = makeGrid(dim)
55      global openFeilds
56      point = randomStart(dim)
57      print("First move "+ str(point))
58      while (len(openFeilds) < (dim*dim)):
59
```

As seen in line 56 we select a random point to start with and build our knowledge from there.

Now after our first move the updateGrid will change from all points showing “x” to as follows:

First move (0, 1):

[0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 1, 1, 1, 0]

[1, 1, 1, 1, 'x', 1, 0]

['x', 'x', 1, 1, 1, 1, 0]

['x', 2, 1, 0, 0, 0, 0]

[1, 1, 0, 0, 0, 1, 1]

[0, 0, 0, 0, 0, 1, 'x']

Which opens up all the connected fields with weight 0. Now the knowledge base has something to work with. The working of which will be discussed under the next heading.

1.1.4 Knowledge Base:

We used a key value pair dictionary to implement the knowledge base as follows:

```python
def knowledgeBase(dim):

    global knowledgeBaseList
    global flagCount
    global flagedFeilds

    #initializing the knowledge base
    if not knowledgeBaseList:
        for i in range(dim):
            for j in range(dim):
                knowledgeBaseList.__setitem__((i,j),0)
```

Initially we start out with the assumption that all the fields are empty. With analogous the real world approach when no information is known to us. Hence we set the probability of any field having a mine to Zero.

But after the first move is made we again call the knowledgeBase () to update the probability of the fields. Which looks something like this (it is to be kept in mind that the probabilities here are a cumulative of all past iterations. Hence is more of a reference number and can exceed 1 after certain iterations. But that is not a problem because we flag only the ones with the highest probability).

```
▼ knowledgeBaseList = {dict} <class 'dict'>: {(0, 0): 0.8333333333333333, (0, 1): 1.0333333333333332,
    (0, 0) (140337346981192) = {float} 0.8333333333333333
    (0, 1) (140337346981256) = {float} 1.0333333333333332
    (0, 2) (140337346981320) = {float} 0.5333333333333333
    (0, 3) (140337346981384) = {float} 0.2
    (0, 4) (140337346981448) = {int} 0
    (0, 5) (140337346981512) = {int} 0
    (0, 6) (140337346981576) = {int} 0
    (1, 0) (140337346981640) = {int} 0
    (1, 1) (140337346981704) = {int} 0
    (1, 2) (140337346981768) = {int} 0
    (1, 3) (140337346981832) = {float} 0.5333333333333333
    (1, 4) (140337346981960) = {int} 0
    (1, 5) (140337346982024) = {int} 0
    (1, 6) (140337346982088) = {int} 0
    (2, 0) (140337346982152) = {int} 0
```

Now once the knowledgeBase() has been updated we again print the updatedGrid with all the open and flaged field which looks something like this:

printing after flaging:(2, 4)

[0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 1, 1, 1, 0]

[1, 1, 1, 1, 'F', 1, 0]

['x', 'x', 1, 1, 1, 1, 0]

['x', 2, 1, 0, 0, 0, 0]

[1, 1, 0, 0, 0, 1, 1]

[0, 0, 0, 0, 0, 1, 'x']

As we can see above after the first iteration field (2,4) has been flaged. Now in the next iteration point (3,1) is marked as it is the field with the highest probability of having a mine.

printing after flaging:(3, 1)

[0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 1, 1, 1, 0]

[1, 1, 1, 1, 'F', 1, 0]

['x', 'F', 1, 1, 1, 1, 0]

['x', 2, 1, 0, 0, 0, 0]

[1, 1, 0, 0, 0, 1, 1]

[0, 0, 0, 0, 0, 1, 'x']

Similarly in the consecutive iterations all the points with highest probability are marked. We use up all the flags first without making another move.

printing after flaging:(6, 6)

[0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 1, 1, 1, 0]

[1, 1, 1, 1, 'F', 1, 0]

['x', 'F', 1, 1, 1, 1, 0]

['x', 2, 1, 0, 0, 0, 0]

[1, 1, 0, 0, 0, 1, 1]

[0, 0, 0, 0, 0, 1, 'F']

printing after flaging:(4, 0)

[0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 1, 1, 1, 0]

[1, 1, 1, 1, 'F', 1, 0]

['x', 'F', 1, 1, 1, 1, 0]

['F', 2, 1, 0, 0, 0, 0]

[1, 1, 0, 0, 0, 1, 1]

[0, 0, 0, 0, 0, 1, 'F']

Once all the flags are marked the knowledgeBase() then returns the field with the least probability as the next field to move to. The set returned from does not contain the already opened fields or the ones flaged. This can be seen in the following code:

```
count = len(knowledgeBaseList)
fPoint = ()
temp = 5000

for tempPoint in setup_cells(dim):

    val = knowledgeBaseList.get(tempPoint)
    if (tempPoint not in openFeilds and tempPoint not in flagedFeilds and temp>val):
        temp = val
        fPoint = tempPoint

return fPoint
```

In the above code snippet we can see how we find the field with the least likely hood of having a bomb(To be kept in mind sometimes multiple field might have the same likely hood, in which the choice is as good as a random guess. Which again is analogous to the real world scenario, this in some cases can result in the loss of the game).

Now once the knowledgeBase() flags the most likely fields and returns the next field to move to we open up the next field as follows:

```
58  while (len(openFeilds) < (dim*dim)):
59
60      row = point[0]
61      col = point[1]
62
63      if(referenceGrid[row][col].bomb == False):
64          if(referenceGrid[row][col].weight == 0):
65              updatedGrid[row][col].open = True
66              openFeilds.append((row,col))
67              neighbour = neighbours(point,dim)
68
69              for temp in neighbour:
70                  openFeilds.append(temp)
71
72              for temp in neighbour:
73                  if (referenceGrid[temp[0]][temp[1]].weight == 0):
74                      tempNeighbour = neighbours(temp,dim)
75                      for tempTemp in tempNeighbour:
76                          neighbour.append(tempTemp)
77                          openFeilds.append(tempTemp)
78              #print(openFeilds)
79
80          else:
81              updatedGrid[row][col].open = True
82              openFeilds.append((row,col))
83      else:
84          print("Dead")
85          return
```

Here we can see how we check if the next field has a bomb or not. If it has a bomb we die else we check the weight of the field and update it. If the weight of the field is Zero we open all the neighboring fields as we would in a real world scenario.

Next movie to: (3, 0)

[0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 1, 1, 1, 0]

[1, 1, 1, 1, 'F', 1, 0]

[2, 'F', 1, 1, 1, 1, 0]

['F', 2, 1, 0, 0, 0, 0]

[1, 1, 0, 0, 0, 1, 1]

[0, 0, 0, 0, 0, 1, 'F']

Now that all the fields are open and as in line 58 len(openFields) = (dim*dim)(implies number of open fields equals the total number of fields). We now print that we have won the game.

Printing updated grid

[0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 1, 1, 1, 0]

[1, 1, 1, 1, 'F', 1, 0]

[2, 'F', 1, 1, 1, 1, 0]

['F', 2, 1, 0, 0, 0, 0]

[1, 1, 0, 0, 0, 1, 1]

[0, 0, 0, 0, 0, 1, 'F']

you won!

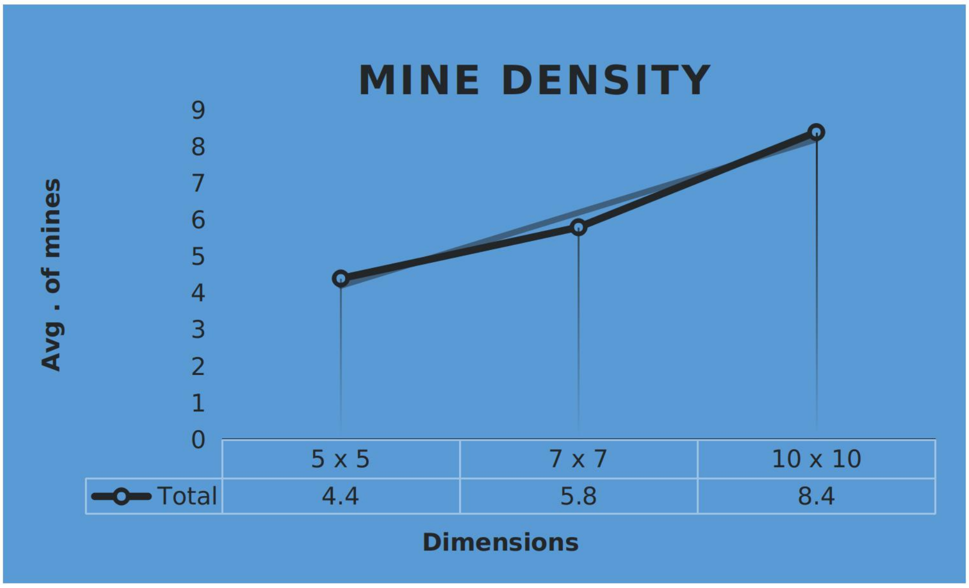
Performance: For a reasonably-sized board and a reasonable number of mines, include a play-by-play progression to completion or loss. Are there any points where your program makes a decision that you don't agree with? Are there any points where your program made a decision that surprised you? Why was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

[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 1, 1, 0]
[0, 0, 0, 1, 'F', 1, 0]
[0, 0, 0, 1, 2, 2, 1]
[0, 0, 0, 0, 1, 'F', 'F']
[0, 0, 0, 1, 2, 'x', 'x']
[0, 0, 0, 1, 'F', 'x', 'x'

In the above graph we can see that our program acted in a way that we don't want him to act as the probability to be a bombs at (4,6) and (5,5) is same but it put the flag at (4,6) but not in (5,5).

The reason our program is not flagging that position because we are using greater than function instead of using equal to. So it first get (4,6) and flagged it and comes out of the loop and didn't flag (5,5)

Performance: For a fixed, reasonable size of board, plot as a function of mine density the average final score (safely identified mines / total mines). This will require solving multiple random boards at a given density of mines to get good average score results. Does the graph make sense / agree with your intuition? When does minesweeper become 'hard'?



Dimension	Mines	1	2	3	4	5	average
5 x 5	5	2	5	5	5	5	4.4
7 x 7	7	4	7	7	7	4	5.8
10 x 10	10	9	8	8	7	10	8.4

As we can see from the graph that in 5 x 5 board there are total number of 5 mines (10%). We did 5 iterations for each dimension. The average number of mines which our program can able to find the mines for 5 x5 is 4.4 and for 7 x 7where there are total number of 7 mines is 5.8 and for 10 x 10 its 8.4 . Yes, the graph makes sense as the number of dimension of board increases out program is finding difficulty to find the proper place of bombs.

The minesweeper becomes hard when we increase the number of mines in a fixed dimension then the probability of appearing 0 and the maximum number will be less and our knowledge base gets difficulty to find the correct position of bomb as our knowledge base is populated with positions having so many small values.

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?

The time complexity for the algorithm is O(n^3). It is problem specific as we want to improve the accuracy of the algorithm or we can say we want to make our algorithm so that it can win every time and therefore we compromised on the time complexity part.

The space complexity for our algorithm is O(n^2). It is implementation specific constraint as the worst case is n^2 we can improve this by deleting list which helps us so save some space.

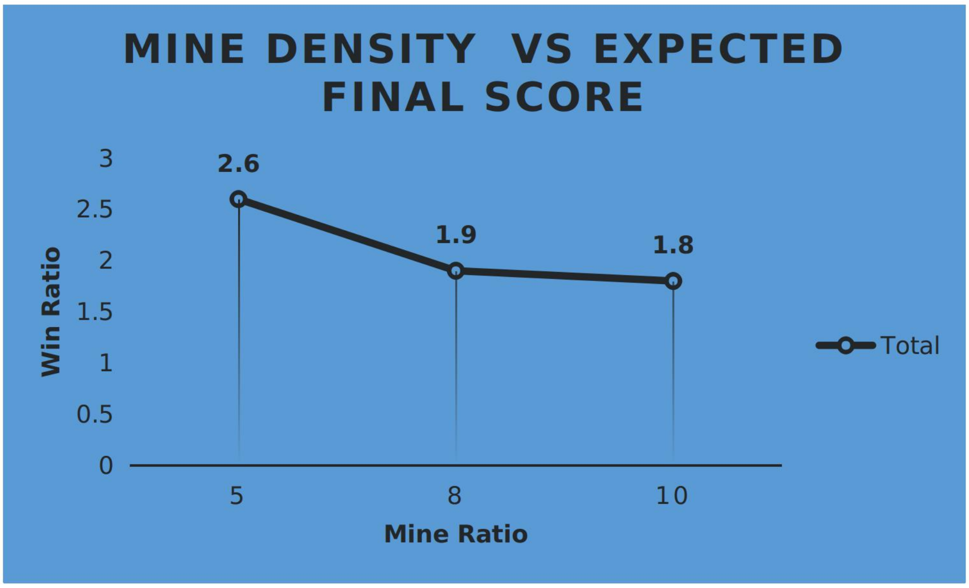
Improvements: Consider augmenting your program's knowledge in the following way - tell the agent in advance how many mines there are in the environment. How can this information be modeled and included in your program, and used to inform action? How can you use this information to effectively improve the performance of your program, particularly in terms of the number of mines it can effectively solve? Re-generate the plot of mine density vs expected final score, when utilizing this extra information

In our implementation of this concept we equated the number of the flags to the number of bombs. With this knowledge in hand we modelled the algorithm to continuously keep track of the fields with the highest probability of having a bomb and flagging them.

Now the knowledgeBase() will return the next most likely safe point and this would open up other fields. With this new-found information we will run the flagging algorithm again to update the flags to the next most likely fields with bombs.

This relation between bombs and flags improves the efficiency of the program and also simplifies the code to great degree compared to the code required to guess the number of bombs and flag random number point. In the below table we made 5 x 5 , 7 x 7 , 11 x 11 board and implemented the algorithm with different mine density of 5 % ,8 5 & 10 % on each matrix of board and get the average value of final score. From the below graph we can conclude that as the mine desity increases the probability of win ratio decreases.

dim	Mine Ratio	1	2	3	4	5	6	7	8	9	10	Average
5 x 5	5	1	1	1	1	1	1	1	1	1	1	100%
	8	1	1	1	1	1	1	1	1	1	1	100%
	10	1	0	1	1	1	1	1	1	1	1	90%
7 x 7	5	1	1	1	0	0	1	1	0	1	1	70%
	8	1	1	1	0	1	1	1	0	1	0	70%
	10	0	0	1	0	1	1	1	1	1	0	60%
11 x 11	5	1	1	1	1	1	1	1	1	1	0	90%
	8	0	0	0	0	1	0	0	0	0	1	20%
	10	1	0	0	1	1	0	0	0	0	0	30%



4.Chains of Influence

4.1 Based on your model and implementation, how can you characterize and build this chain of influence?

In our model of implementation implementing the minesweeper with a AI we went with the following approach:

1. The first point chosen is random and if this point has a weight(adjacent bombs) there will be no chain reaction and no further fields will be minded.

2. If the first field chosen has a weight of 0 then all the eight or so fields surrounding it will be minded. Their “open” property will be set to True.

```
56      point = randomStart(dim)
57      print("First move "+ str(point))
58      while (len(openFeilds) < (dim*dim)):
59
60          row = point[0]
61          col = point[1]
62
63          if(referenceGrid[row][col].bomb == False):
64              if(referenceGrid[row][col].weight == 0):
65                  updatedGrid[row][col].open = True
66                  openFeilds.append((row,col))
67                  neighbour = neighbours(point,dim)
68
69                  for temp in neighbour:
70                      openFeilds.append(temp)
71
72                  for temp in neighbour:
73                      if (referenceGrid[temp[0]][temp[1]].weight == 0):
74                          tempNeighbour = neighbours(temp,dim)
75                          for tempTemp in tempNeighbour:
76                              neighbour.append(tempTemp)
77                              openFeilds.append(tempTemp)
78                  #print(openFeilds)
79
80              else:
81                  updatedGrid[row][col].open = True
82                  openFeilds.append((row,col))
```

In the above code snippet we can see at line 56 how the first point opened is chosen at random, following this a cascade of events takes place if the weight of point opened is zero. Where all the neighboring points and opened and same will happen to all the zeros in the neighboring points as well. This is modeled after the real world scenario.

3. Now we call the knowledgeBase(). The knowledgeBase() then updates all the fields with theirs corresponding probability of having bombs(we store this data in the form of a dictionary).

```
80             else:
81                 updatedGrid[row][col].open = True
82                 openFeilds.append((row,col))
83         else:
84             print("Dead")
85             return
86 
87         #printing updated maze
88         for x in openFeilds:
89             updatedGrid[x[0]][x[1]].weight = referenceGrid[x[0]][x[1]].weight
90             updatedGrid[x[0]][x[1]].open = True
91 
92         #printing the maze
93         printGrid(updatedGrid,dim)
94 
95         point = knowledgeBase(dim)
96         print("printing updated maze:")
97 
98         printGrid(updatedGrid,dim)
99 
```

As we can see in line 95, once the knowledgeBase() updates all the probabilities it marks all the point with the highest probability of having bombs with flags and then returns the point with the least probability of having a bomb.

4.The flagging process in the knowledgeBase() can be seen the code snippet below:

```
    for tempo in flagedFeilds:
        i=tempo[0]
        j=tempo[1]
        openFeilds.remove((i, j))
        updatedGrid[i][j].flag = False
        updatedGrid[i][j].bomb = False
        updatedGrid[i][j].open = False
    flagedFeilds.clear()
```

```
    while True:
        tempList=0.00
        temp = ()
        for i in range(dim):
            for j in range(dim):
                if((knowledgeBaseList.get((i,j))*10.00) >= tempList and (i,j) not in
                    print(knowledgeBaseList.get((i,j))*10.00)
                    temp = (i,j)
                    tempList = (knowledgeBaseList.get((i,j))*10.00)
        if not(not temp):
            flagedFeilds.append((temp[0],temp[1]))
            openFeilds.append((temp[0],temp[1]))
            updatedGrid[temp[0]][temp[1]].open = True
            updatedGrid[temp[0]][temp[1]].flag = True
            updatedGrid[temp[0]][temp[1]].bomb = True
            print("printing after flaging:"+str(temp))
            printGrid(updatedGrid,dim)
        flagCount-=1
        if flagCount<=0:
            break
```

We first start by clearing all the flags assigned in the previous iteration(seen in line 136) we when recompute the probability with the newly opened field from the previous iteration and post this we again reassign the flags to then highest weighed fields.

5.Finally like in the real world scenario if we hit a road block and no further moves are available. But the number of mined fields do not equal total number of fields we perform a random restart. This will pick a random point from the remaining points.

6.The game ends once we have either mined or flagged all the available fields. This can be seen in line 58 of the code.

```
55        global openFeilds
56        point = randomStart(dim)
57        print("First move "+ str(point))
58        while (len(openFeilds) < (dim*dim)):
```

5.Bonus:Dealing with Uncertainty

5.1 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.

To make a case that when we select any index the information about the neighbors should display after some time we used multi-threading process in which we made two threads one in function which returns the information about the neighbors and another in the knowledgebase function. This will affect our algorithm in a way that if we select any index and we don't get information about the neighbors that this cell will again be open by our algorithm which will increase time to solve the board.

5.2 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.

The problem statement is how do we tackle the situation when we select one node and the number which appears is less than the actual probability of mines in the neighbors. To overcome this problem what we do is to open a node which isn't open yet without opening the neighboring element of the opened node because the risk is high if we open the neighboring node as the probability of mine shown is not correct. So we open another node and this will help in increasing the number shown (probability of mine) and hence give us some idea which node to open next.in the form of a dictionary).

5.3 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

To overcome this problem we can use same approach as we used in previous question.