CS 520: Assignment 2

Minesweeper, Inference-Informed Action

Bharath Reddy

Brp97

191003117

Ishaan Singh

is393

191003073

3. Questions and Writeup

Game Modeling:

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. Flagged

4. A closed field can be opened or flagged.

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 without any mine then you win.

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

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

```
25  #every element of the grid template:
26  class gridElement:
27      def __init__(self):
28          self.location = ()
29          self.bomb = False
30          self.weight = 0
31          self.flag = False
32          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 flagged or not.

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

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

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 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.

3.2 Inference: Collecting a new clue and modeling/processing of new data. Updating the knowledge base based on a new clue. Decisions: Given a current state of the board, and a state of knowledge about the board, how does you program decide which cell to search next? Also lets check the risks and how to face them.

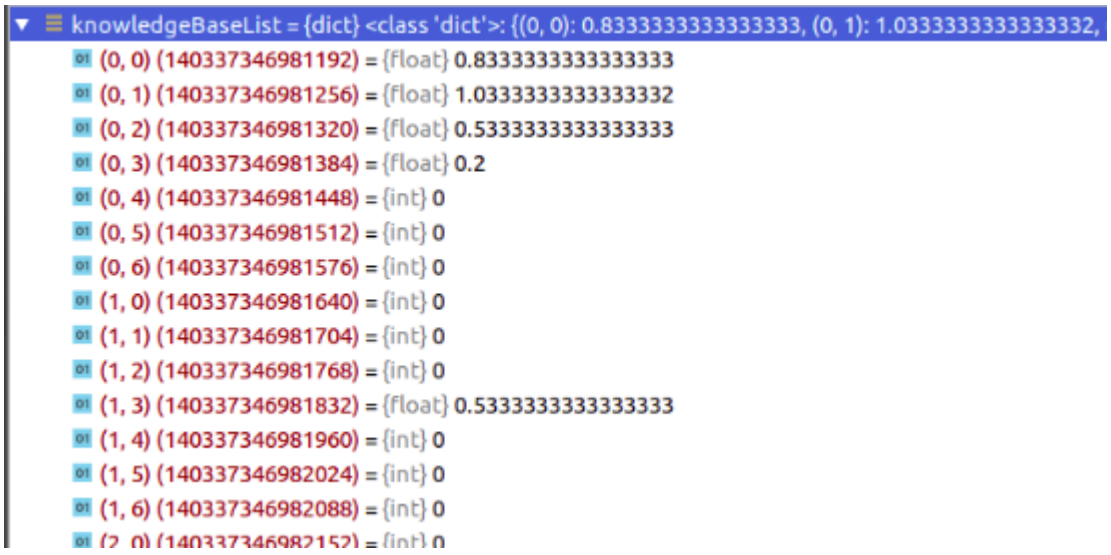
3.3 Decisions: Given a current state of the board, and a state of knowledge about the board, how does your program decide which cell to search next? Are there any risks, and how do you face them?

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

```
117     def knowledgeBase(dim):
118
119         global knowledgeBaseList
120         global flagCount
121         global flagedFeilds
122
123         #initializing the knowledge base
124         if not knowledgeBaseList:
125             for i in range(dim):
126                 for j in range(dim):
127                     knowledgeBaseList.__setitem__((i,j),0)
128
```

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).



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

printing after flagging:(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 flagged. 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 flagging:(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 flagging:(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 flagging:(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 flagged. This can be seen in the following code:

```
185     count = len(knowledgeBaseList)
186     fPoint = ()
187     temp = 5000
188
189     for tempPoint in setup_cells(dim):
190
191         val = knowledgeBaseList.get(tempPoint)
192         if (tempPoint not in openFeilds and tempPoint not in flagedFeilds and temp>val):
193             temp = val
194             fPoint = tempPoint
195
196     return fPoint
197
```

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!

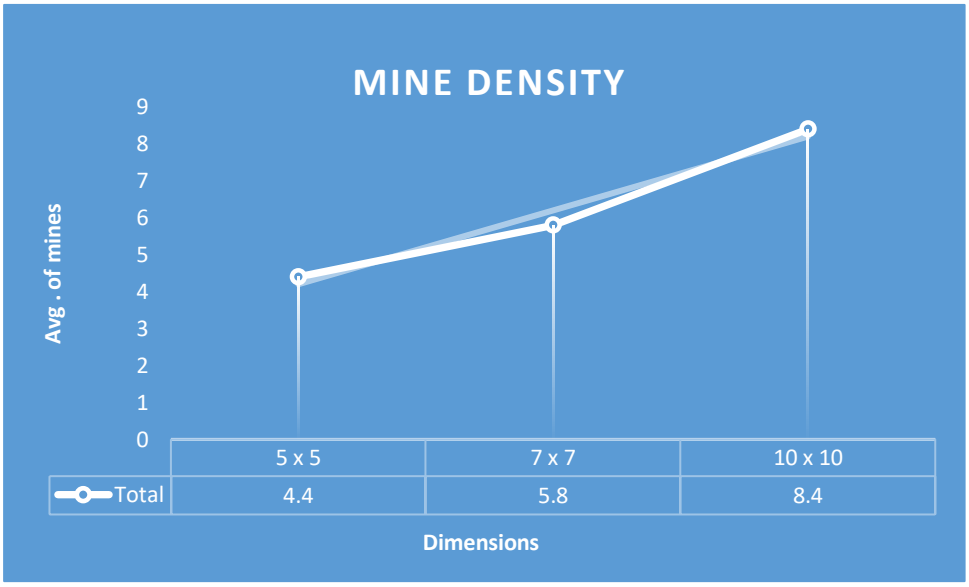
3.4 Performance: For a reasonably-sized board and a reasonable number of mines, include a play-by-play progression to completion or loss. Are there any points where your program makes a decision that you don't agree with? Are there any points where your program made a decision that surprised you? Why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

3.5 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 dimensions 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.

3.6 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.

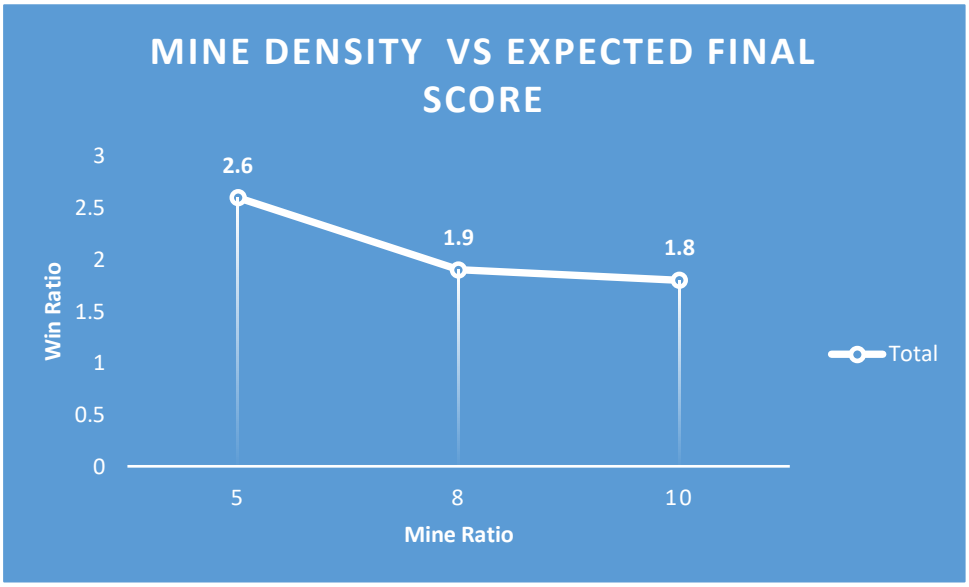
3.7 Improvements: Consider augmenting your program's knowledge in the following way - tell the agent in advance how many mines there are in the environment. How can this information be modeled and included in your program, and used to inform action? How can you use this information to effectively improve 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 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 density 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%



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:

```
136     for tempo in flagedFeilds:
137         i=tempo[0]
138         j=tempo[1]
139         openFeilds.remove((i, j))
140         updatedGrid[i][j].flag = False
141         updatedGrid[i][j].bomb = False
142         updatedGrid[i][j].open = False
143     flagedFeilds.clear()
```

```
171     while True:
172         tempList=0.00
173         temp = ()
174         for i in range(dim):
175             for j in range(dim):
176                 if((knowledgeBaseList.get((i,j))*10.00) >= tempList and (i,j) not in
177                     print(knowledgeBaseList.get((i,j))*10.00)
178                     temp = (i,j)
179                     tempList = (knowledgeBaseList.get((i,j))*10.00)
180         if not(not temp):
181             flagedFeilds.append((temp[0],temp[1]))
182             openFeilds.append((temp[0],temp[1]))
183             updatedGrid[temp[0]][temp[1]].open = True
184             updatedGrid[temp[0]][temp[1]].flag = True
185             updatedGrid[temp[0]][temp[1]].bomb = True
186             print("printing after flaging:"+str(temp))
187             printGrid(updatedGrid,dim)
188         flagCount-=1
189         if flagCount<=0:
190             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)):
```

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

→Usually the largest number of fields that can be mined in a single instance depends upon the number of connected zeros. As when a field with zero weight is mined we automatically mine all the 8 or so adjacent mines and if any other zeros are present in these 8 or so fields then they to will be mined. This is based on the premise that all the fields around the fields with zero weight are empty and do not have bombs.

4.3 How does the length of the chain of influence the efficiency of your solver?

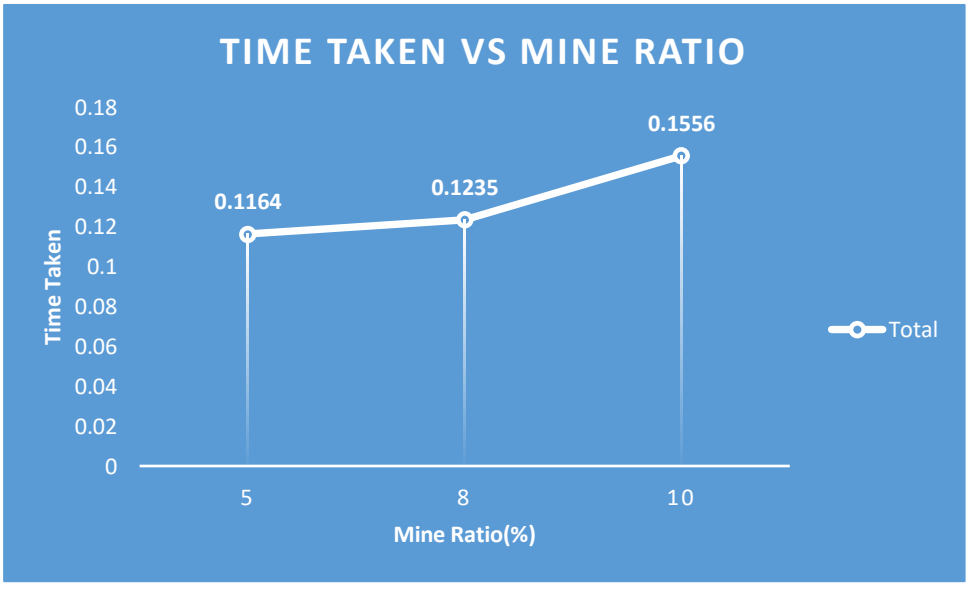
→The longer the length of the chain of influence the more the number of fields that will be mined in a single mining or iteration. This will significantly influence the time for solving the grid as we will have less number of fields to mine if the chain of influence is longer also this will give our knowledgeBase() significant amount of data to work with. The time required to solve is

inversely proportional to the length of the chain of influence.

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

→ To find a board with long chains of reference we have to make the ratio of mines less. By doing this our knowledge base increases and we have to open more indexes to get better information about the indexes which have mines. So, number of mines are inversely proportional to the chain of influence. We did one experiment in which we made 7 x 7 matrix and adjust the mine ratio from 5 % -8%-10 %. When we applied mine ratio of greater than 10 % the probability of winning game became ow around 0.2 . The results from the experiment are in table and plotted in a graph.

Ratio of Mine(%)	1	2	3	4	5	6	7	8	9	10	AVG Time Taken
5	0.217	0.154	0.174	0.115	0.104	0.059	0.094	0.041	0.043	0.163	0.1164
8	0.208	0.164	0.184	0.116	0.145	0.062	0.01	0.088	0.01	0.248	0.1235
10	0.187	0.059	0.256	0.177	0.147	0.145	0.116	0.063	0.118	0.288	0.1556



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

→Theoretically, the longest influence of a given cell can travel is throughout that grid. For example, let's consider the instance where the grid has no bombs, in this case mining a field will result in a chain reaction which results in the mining of the entire grid.

In other cases where there are bombs in the grid the length of influence depends on the number of bombs. Generally, the two share an inverse relationship.

4.6 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?

→ To solve this board more efficiently, yes we can you this notion of minimizing the length of chains of influence as we can select those neighbouring indexes for which we have information in our knowledge base rather than randomly select any indexes.

4.7 Is solving minesweeper hard?

→ It depends upon the dimension of the maze & ratio of bombs if the dimension of the board is high and the ratio of bombs are more than we have to select more random indexes to accurately predict the places where the mines are and if the ratio of bombs are large than the algorithm will run fast and lead to win more efficiently because the information in the knowledge base is adequate to know the position of mines.

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.

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.