Andrew Hoyle
Nicholas Young
Yiming Zhao

Minesweeper Project

Most people's exposure to the game known as Minesweeper is at a very young age. It seems a young child's first instinct when exploring a new computer system or operating system is to search for the games available to play. Since Windows has enjoyed a 90% market share or above on home computing systems since the early 90's almost every child has been exposed to Minesweeper. At first the game seems quite easy to win, but as you advance to the harder and harder difficulties the game seems to get extremely hard quickly, some, would say even **exponentially** so. The following is an in depth look into what the cause of this perceived phenomenon.

The Game

The Minesweeper game is a single player game that in which the object of the game is to clear an abstract minefield without detonating a mine that are hidden under the tiles. The player clicks on the board to a reveal a tile. There are then two possibilities, either a number is revealed which represents the number of adjacent mines to that tile, or the player dies because that specific tile contained a mine. The overall goal of Minesweeper is to reveal all tiles on the field that do not contain a mine. You may also mark each tile that you believe contains a mine, but this is not a necessary win condition to the game.

Now that we have an understanding of the game and how it is played we then posed a question. Is it possible Given a rectangular grid of partially marked tiles with numbers and and/or mines, some squares being left blank, to determine if there is some pattern of mines in the blank squares that give rise to the numbers seen? In more simple terms, to determine whether the data given to you is consistent. To put it more simply, is there an accurate and efficient algorithm that exists that can solve the Minesweeper consistency problem in polynomial time? The first step in our problem was to figure out how computationally complex Minesweeper actually is. In the Windows XP version of Minesweeper in Advanced mode the player is presented with a board that has 480 squares and 99 mines. When looking at the possible combinations you can see that this gives you about 10^{100} possible ways the mines can be configured on the playing board. It is readily apparent at this point that any algorithm attempting to compute all of these combinations would not take polynomial time. So how do we go about tackling something this challenging?

Its Creation

First we will take a little look at Minesweeper History to give us a better perspective of the problem. The first variations of Minesweeper had its origins on large data mainframes of the 1960's and 1970's. The first versions of Minesweeper were simpler versions of the Minesweeper game we are familiar with today. A popular predecessor to Minesweeper was RLogic created by Conway, Hong and Smith that was available for MS-DOS as early as 1985. RLogic is much simpler as it consists of commanding a US Marine that must make his way across an imaginary minefield to his/her objective. This imposed condition reduces the complexity drastically. The version we are most familiar with today was originally created by Curt Johnson for the OS/2 operating system. It was then ported over to Windows 3.1 by both Johnson and Robert Donner. It has been included with every subsequent release of Windows. There are however various versions of Minesweeper that span multiple operating systems that vary in size, method of play, and complexity.

Measuring Complexity

In 2000 Mathematician Richard Kaye was able to prove that the Minesweeper consistency problem is in fact NP-Complete in his article in the *Mathematical Intelligencer vol. 22 no 2* "Minesweeper is NP-Complete". How was he able to show this? Kaye was able to show that that the SAT problem is reducible to the Minesweeper consistency problem, therefore the Minesweeper consistency problem is no harder than SAT. Kaye was able to show that it is possible to simulate any sort of logic circuit using the Minesweeper playing board. As shown in the attached presentation we are able to simulate AND and OR gates, Wires, Crossovers and Splitters respectively. Therefore any algorithm solving the Minesweeper consistency problem would allow us to solve questions about simple (propositional) logic such as SAT. Since SAT is NP-Complete then the Minesweeper problem is also a member of NP-Complete. That is Minesweeper is NP-Complete to determine whether a given grid of uncovered, correctly flagged and unknown squares, the labels of the foremost given , has an arrangement of mines for which it is possible within the rules of the game. Membership in NP is established by the arrangement of the mines as a certificate.

Implementation of Board and Game

Tile Struct

The tiles have data members that represent their states in the game. A "mine" bool determines whether the tile has a mine contained at its coordinates. "Discovered" denotes that the tile has been clicked on, and is used to limit the algorithms from cheating as well as cue the display functions whether to print the number of adjacent mines, or a "+" that signifies the tile is unknown. In addition to this, the tiles may be flagged or given a question mark. The question mark is of no use to the algorithms created, it is just present to allow the user to play for future implementations.

Board Class

The tiles know nothing about their neighbors; this is taken care of by the board class. All of the functions of the board class involve randomly creating the tiles, checking to see if the algorithm has won or died, and updating the tiles values when they've been clicked on.

First off, the constructor(mineCount, rows, cols) takes the dimensions and the mine count and stuffs tiles into a 2D vector. Then, random tiles are selected to become the mines until the number of mines on the board matches what was sent to the constructor. A function then runs over all of the tiles to populate their adjMine number by counting the number of mines in the 8 surrounding tiles. The edge tiles are marked as "discovered" and the "edge" bool is set to true.

Playing the Game

The algorithm and board communicate and play the game by passing row and column values by reference every turn. After the algorithm selects a current row, column and enum clickType{click, flag, question}, the deathCheck() function takes a look at the algorithms' selections to see if there's a mine on the selected click. If not, the updateAllMines() function is allowed to unmask one or more tiles. Finally, the winCheck() counts the number of undiscovered mines and if it is not 0, the game loop continues.

```
while(death==false && win==false)
      displayBoard()
      algorithm(row, col, clickType enum, expansion bool)
      deathCheck(row, col)
      updateTiles(row, col)
      winCheck()
if (death==true) "you dead sucka"
if (win==true) "you win!!!"; ++winCount
```

Recursive Expansion

One interesting function found in the board class was the expansion function that causes multiple tiles to be marked as discovered if a tile is clicked on with no adjacent mines. The best way to create this algorithm was to make it be a recursive function. As the code demonstrates, this function can be likened to some sort of robot (like a Roomba) that must travel from the clicked tile in all 8 directions until it bumps into the "wall" made up of a tile that has an adjacent mine value greater than 0. If it heads in one of the directions and the next tile also has 0 adjacent mines, the function is called recursively with the new coordinates. This concept of traveling in one direction the maximum possible distance makes this function an eight-directional depth first search. The "visited" data member of the tile structs had to be created so this robot knew the tiles it had already visited during this expansion process. Before the visited bool was included in the conditional situations stipulating another recursive call, this function got caught in a clockwise infinite loop.

The Algorithms

Minesweeper, due to its limited information given to the user and its complexity, necessitates guesses from the user often multiple times throughout the course of one game. Right off the bat, for example, the user must select a tile to click on. This click is the only freebie given to the player because it is impossible to blow up on the first click of a new game. After that, the user hopes for an advantageous "expansion" that indicates the 100% or 0% presence of a mine, but there are NO GUARANTEES!

The algorithms that try to beat the game all incorporate guessing at the beginning, but the more sophisticated ones try to minimize the guessing later on and then take an educated guess if there are no certainties. Different risk calculations and bools are used to help decide which part of each algorithm are to be used. This group was able implement four algorithms and categorize them into three different levels of sophistication.

1

The first level is quite trivial and is just a function "dumbDumb" that randomly picks a tile making sure only that is undiscovered… much like throwing darts at board blindfolded.

```
do while(grid[row][col].discovered==true)
      row = random#
      col = random#
```


The “2nd level algorithms” have the dumbDumb function run UNTIL there has been an expansion from one of the random guesses. Each discovered tile has its neighbors inspected and the risk calculation is run hopefully yielding that there is a 0% or 100% chance of a mine in an adjacent tile:

```
//risk calc
tileRiskNum = b[i][j].adjMines – b[i][j].adjFlags
tileRiskDenom = b[i][j].adjUnknowns
tileRisk = tileRiskNum / tileRiskDenom

//choice of click
if tileRisk==0
      click it!
if tileRisk==1
      flag-click it!
```

Picture a 3x3 cookie cutter centering on every single discovered tile on the board, and using the risk calculation equation on the 8 surrounding tiles.

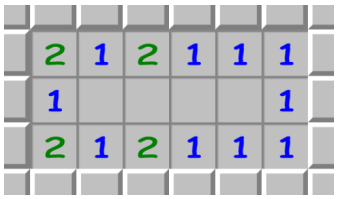
So what do the 2nd level algos do if they’ve not been fortuitous with a generous sea expansion and now have no guarantees? We implemented two varying algorithms. The first, called “backToDumb,” sets the “expansion” bool back to false, causing the dumbDumb random function to take the helm until another expansion.

```
//backToDumb
if expansion==false
      dumbDumb()       //nested level 1
cookieCutter()         //checks for 0% or 100% risk
if risk!=0 || 1
      expansion=false  //reset it to false so dumbDumb is called
      dumbDumb()       //level 1 called again
```

The other slightly more successful algorithm, called “calcRisks” once again looks at the discovered tiles, saves their row and column values, the risk of a mine being around, and makes a minimum priority queue. The minimum risk is then clicked on.

```
//calcRisks()
if expansion==false
      dumbDumb()       //nested level 1
cookieCutter()         //checks for 0% or 100% risk
if risk!=0 || 1        //no guarantees of mine or flag...
      PQ of tiles’ risks
      educated guess
if PQ.empty()          //rare case where PQ is empty
      dumbDumb()       //gotta guess again!
```

This calcRisks algorithm has a weak spot, however, and the most successful 2nd level algorithm may be a combination of these two using one final risk calculation. Picture the following scenario:



This scenario after the first click of the game displays a tiny expansion that is skimping on its hints to the algorithm. Since there are no guarantees here (at least from only looking at one tile's risk at a time), the PQ part of calcRisks has now activated and only is concerning itself with the adjacent tiles of those on the edge of the expansion. The minimum risks of 20% are found on the right corners. Depending on the difficulty setting, picking a random tile like dumbDumb would have a slightly better chance than one of the tiles adjacent to the two right corners:

- Beginner 10 [mines] / (81 [total tiles] – 18 [uncovered tiles]) = 15.8% risk!
- Intermediate 40 [mines] / (256 [total tiles] – 18 [uncovered tiles]) = 16.8% risk!
- Advanced 99 [mines] / (81 [total tiles] – 18 [uncovered tiles]) = 21.4% risk

The calcRisks algorithm is at a disadvantage if it is playing intermediate or beginner in scenarios like this. Populating the PQ with all of the risks and then one final risk taking into account all of the tiles and remaining mines may help assuage the risks brought up in a measly initial expansion. This most sophisticated PQ remains to be implemented.

3

The 3rd and final algorithm, called "2atATime," takes the calcRisks algorithm and wedges another sub function between the cookieCutter 0% and 100% function and the PQ function.

```
//2atATime()
if expansion==false
      dumbDumb() //nested level 1
cookieCutter()         //checks for 0% or 100% risk
if risk!=0 || 1        //no guarantees of mine or flag...
      PQ of tiles' risks
      educated guess
lookAt2Tiles()         //new part for 3rd level
if PQ.empty()          //rare case where PQ is empty
      dumbDumb()       //gotta guess again!
```

Instead of the function looking at one tile at a time, it looks at the overlapping undiscovered tiles between two tiles and makes some comparisons. It essentially finds out if there is a number of mines that MUST be encapsulated in the tiles' overlapping adjacent tiles, and incorporates this known number into the risk calculations of the tiles surrounding tiles NOT overlapping. If these comparisons don't yield any guarantees, the PQ section picks up again and makes an educated guess.

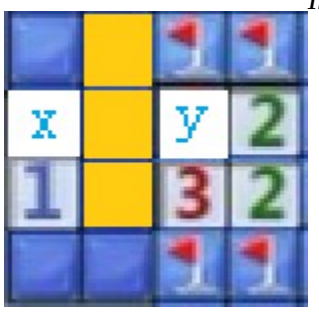
Level 3 Sub-Functions

Inner analyze:

When Flag(x)<Mine(x)<Flag(x)+Unknown(x), then we know the number of mines around x is Mine(x)-Flag(x), but the exact location is still unknown. We can start by analyzing the inner grids. If the location of the mines is unknown then you must choose the tile with the lowest risk.

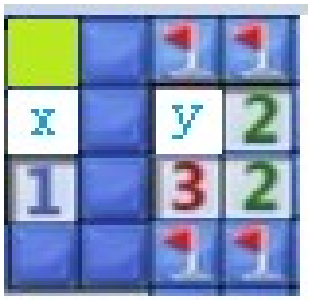
If there are two grids x and y which are in similar configurations but the mines between them are unknown then we are forced to look at the inner grid. We can break down this process in to 5 parts.

Inner(x, y):



First step is to identify the inner grid of tile x and y. These are shown in yellow above.

Outer(x, y):



The second step is to identify the outer grids of tile x and y. We only mark tiles that we are not 100% certain about.

Outer(y,x):

We now have to have to try to identify the outer grids of tile y and x. We can see from above that this total is zero.

Mine(x):

We will then need to find the number of mines around the x grid. From the above example that this number is two.

Mine(y):

Next we will need to find the number of mines around the y grid. From the above example we can see that this number is one.

Combining them:

Once we find these numbers we can then use these numbers in two subsequent algorithms. The first algorithm is Min(Inner(x,y), Mine(x), Mine(y)). This first algorithm is to find the maximum number of mines contained in the inner grid. The second algorithm is Max(0, Mine(x)-Outer(x,y), Mine(y)-Outer(y,x)). This second algorithm will then find the minimum number of mines that are contained in the inner grid. Once we have calculated both numbers from these equations above we can use the results to calculate the number of mines. If for instance, the minimum number calculated equals the max number calculated then we know for certain the number of mines contained in the inner grid which will be the minimum. Once this number is obtained we can click safe tiles and flag mines as needed. We can then run this algorithm continuously until we come across a situation where our result is not 100% certain. We will then have to use the risk rate analysis method and choose the lowest risk tile.

```
//modified tileRisk function after level 3 overlap analysis
tileRiskNum = b[i][j].adjMines - b[i][j].adjFlags - (min=max_number)
tileRiskDenom = b[i][j].adjUnknowns - (number_of_tiles_in_inner)
tileRisk = tileRiskNum / tileRiskDenom
```

Improved PQ

There are some inevitable heart-breaking scenarios where the user is confronted with no choice but to guess on the last few remaining tiles. All the work done by 2atATimeup to this ending "coin toss" is fruitless if the guess is incorrect. ;-(

```
//output and explosion from last 2-tile analysis
X: (16, 20)
Y: (16, 18)
        outerX before: (15, 19); (16, 19)
```

```
        outerY before: (15, 19); (16, 19)
        outerX AFTER:
        outerY AFTER:
        innerXY AFTER: (15, 19); (16, 19)
deuces wild!, you gotta guess again, no 100% or 0%
PQ now has tileRisk around: 15, 18: 0.5
PQ now has tileRisk around: 15, 20: 0.5
PQ now has tileRisk around: 16, 18: 0.5
PQ now has tileRisk around: 16, 20: 0.5
PQ has picked: 15, 19 with risk of: 0.5
You Blew Up at: 15, 19
                    game view:
                  ################################
                  #         1F1      1F21 2F32FF21#
                  #         111      12F1 2F3F322F#
                  # 111              1221 11211 11#
                  # 1F1111111        1F211        #
                  # 1111F11F1       1222F1      11#
                  #    111111       1F1111      1F#
                  #                 11211   111 11#
                  #11  1111221      123F1   1F1   #
                  #F1  1F22FF1    112FF21   111 11#
                  #11  112F321    1F2221        1F#
                  #     1221      111           11#
                  #     1F1  11211           111 #
                  #     222  1F2F1 12321     1F1 #
                  #     1F1  11211 1FFF1     111 #
                  #     2331111111 13+31         #
                  #     1FF11F11F1  1+1          #
                  ################################
                        behind the covers:
                  ################################
                  #         1*1      1*21 2*32**21#
                  #         111      12*1 2*3*322*#
                  # 111              1221 11211 11#
                  # 1*1111111        1*211        #
                  # 1111*11*1       1222*1      11#
                  #    111111       1*1111      1*#
                  #                 11211   111 11#
                  #11  1111221      123*1   1*1   #
                  #*1  1*22**1    112**21   111 11#
                  #11  112*321    1*2221        1*#
                  #     1221      111           11#
                  #     1*1  11211           111 #
                  #     222  1*2*1 12321     1*1 #
                  #     1*1  11211 1***1     111 #
                  #     2331111111 13*31         #
                  #     1**11*11*1  111          #
                  ################################
```

The game has no sympathy and only rewards the users if they don't blow up. This “bottleneck” of risk is a very humbling situation for the advanced algorithms after all their hard work. Sometimes, however, there are scenarios toward the end of the game where another algorithm - that we'll categorize on the 4th level - could snag a few more victories.

This newest algorithm gets its own rank because it is cognizant of the remaining number of mines in addition to the discovered tiles' surrounding risk percentages. A new sub-function is appended to the 3rd level algorithm just before the PQ and only activates when the risk calculation looking at two tiles at a time yields no guarantees.

```
//Level 4
dumbDumb()
CookieCutter()          //one at a time
lookAtTwoTiles()        //two at a time
Hypotheticals()         //multiple scenarios 4th level function
PQ()
dumbDumb()
```

As the undiscovered mines become few and there are no guarantees found, the Hypotheticals function activates looking at all possible scenarios. It has the ability to ask itself “If a mine were placed here, then that would mean a mine couldn't be here...” etc. Multiple game scenarios would branch out from this point, one for each undiscovered tile being treated as a mine. If there are ambiguities within these hypothetical scenarios, there would be another level of hypothetical branches! Every game scenario would have to be saved for comparison, but if there is a unique number of hypothetical flags in a scenario, and that matches the number of actual remaining mines, then the game is solved. Slide 67 in the attached presentation has two different possible solutions with unique mine counts: 6 and 7. All algorithms except level 4 must guess because they have no knowledge of the remaining mines!

Running Times

As the levels of intelligence build off of one another and become more mindful of multiple elements of the game - like one or two tiles' adjacent risks and the number of remaining mines on the board - the running time becomes more expensive very quickly. The cookieCutter algorithm runs in O(n) time where n=number of tiles. The 2atATime runs in O(n^2) time because the nested loop of the second tile being compared to the first. This hypothetical algorithm would be MUCH more costly because it has to play the game as many times as there are uncovered tiles! A limiter would have to be placed on the 4th

level's Hypotheticals function to have it only kick into gear toward the end of the game... it would be an efficiency vs. risk balancing act.

Heuristics – refer to *minesweeper_heuristics.pdf*

A wrapper around the playTheGame() function was created, and the tallies of games won by each algorithm were placed into structs after a for-loop played the game 10,000 times. Each of the algorithms played the same randomly generated 10,000 games to keep it fair. A final for-loop was placed around everything to allow the number of mines to be increased in increments of five for displaying the win count vs. number of mines on the advanced, beginning, and intermediate boards.

There are interesting intervals among the three plots that show that the calcRisks 2nd level algorithm beating the 3rd level 2atATime winning, but as the mine counts get closer to the final mine count value, the 3rd level edges out victoriously. The close competition between calcRisks and 2atATime are difficult so see, so the samples where calcRisks is winning have been highlighted in blue on the heuristics spreadsheet. The backToDumb algorithm performed very similarly to the calcRisks algorithm at first, but then petered out more quickly as the mine count increased. Here are the final results for the 3 official game settings and their mine counts:

- beginner with 10 mines: calcRisks: 25%, backToDumb:23%, 2atATime: 27%
- intermediate with 40 mines: calcRisks: 5.31%, backToDumb: 3.51%, 2atATime 7.07%
- advanced with 99 mines: 2atATime: 0.89%, backToDumb: 0.03%, calcRisks 0.29%

Future Work

First off, our group would like to implement the more sophisticated PQ and 4th level algorithm that both take into account the number of remaining mines. Also, it would be interesting to work with a graphics library to make the game function on a GUI where a human user could also play the game. After this, it would be a surprisingly simple jump to make the board class work in 3D by tweaking the constructor, expansion recursive function, and vector<vector<vector<tiles> > > storing the tiles. (The recursive expansion function, for example, would become a 26-directional DFS.) Also, the freebie of the first click must be incorporated into future implementations because right now, the player is able explode

on the first click. Testing location of the initial clicks on the board, (centered, corners, close to corners), and their average sea expansion sizes would be another good variable to isolate.

Conclusion

This was an exciting project to work on because of its complexity of implementation and analysis. The fourth algorithm concept that looks ahead to possible branches of moves is not unlike what computers calculate when playing chess or any strategy game... “If my opponent does this, then I have the best likelihood of winning if I take this choice branch.” The SAT mapping of any circuit was also very interesting to see in the *Mathematical Intelligencer* article. Speaking of chess and other games involving choices that cause a chain reactions of possible ways to win or lose, it seems as though there are many games that could have logic circuits mapped onto them. Viewing video games as a language of boolean operations is an exciting perspective.

SOURCES

- Kaye, Richard. "Minesweeper Is NP-Complete." *Mathematical Intelligencer* 2nd ser. 22.2 (200): 9-15. Print.
- Kaye, Richard. "Minesweeper and NP-completeness." *Minesweeper and NP-completeness*. Brimingham University, 01 Jan. 2000. Web. 09 May 2012. <http://web.mat.bham.ac.uk/R.W.Kaye/minesw/ordmsw.htm>.
- "Minesweeper (video Game)." *Wikipedia*. Wikipedia, 29 May 2008. Web. 9 May 2012. <https://en.wikipedia.org/wiki/Minesweeper_%28computer_game%29>.
- "Minesweeper (Windows)." *Wikipedia*. Wikipedia, 22 Sept. 2007. Web. 9 May 2012. <https://en.wikipedia.org/wiki/Minesweeper_(Windows)>.
- Qian, Huimin, and Shouqiu Yu. "Automatic Minesweeper Algorithm." *Science and Technology Innovation Herald* 2009.31 (2009): 250-51. Print.