Minesweeper Solver

This program is an implementation of the popular computer game Minesweeper. It contains several functions, the main of which being a solver which allows the computer to play the game using a combination of tailor-made algorithms.

Contents

Analysis

Introduction

Minesweeper is a single player puzzle game, first officially released as part of a Microsoft game pack in 1990, although its core gameplay has its origins in some of the earliest computer games from the 1960's and 70's. Microsoft Minesweeper is the best-known version of the game and was included in all standard installations of Windows from Windows 3.1 in 1992 up until Windows Vista in 2007.

This project includes my own implementation of the original game, playable by humans and including all the same functionalities as the original game. It also includes other features such as a stats tracker, board generator and trainer. The main problem the program is designed to solve, however, is the game itself. I attempt to create a program which can read, understand and play through a game of Minesweeper by itself, using several tailor-made algorithms which make use of mathematical and logical reasoning to deduce the nest move to make. The game lends itself well to a computational approach to playing, making it a worthwhile project to attempt.

This program and all its features were requested by and designed for my end user, Chris, who was in constant dialogue with me for the duration of the project.

Videos are used throughout this writeup, which are denoted with [n], any references used are denoted with [n]. These can both be found at the end of the document along with a glossary.

Understanding the Problem

Before any research or analysis can take place, a set of initial requirements and objectives are needed. These will guide my exploration of the problem and how to tackle it

Initial Dialogue

My initial dialogue with my end user, Chris was as follows:

Why do you want this program?

> "I enjoy playing Minesweeper, but feel like I often make silly mistakes, I would like to be able to watch the computer play the game, and learn from it. I would also enjoy being able to compare the computer's ability to mine."

Would you want the computer to play randomly generated games?

> "The ability for the computer to generate and solve random grids would be perfect, as I could watch it play and learn from it. Being able to make my own game for it to solve would also be a desired feature."

After making your own board, would you like it to finish the game, or just show you the next move(s)?

> "Just showing the next moves would be good, as I would then be able to get help when I'm stuck, but still be able to continue the game after the computer shows me the next move. I would also like to be able to generate any board I like for it to play."

Would that be for both randomly generated and user generated boards?

"Yes, I would like to be able to specify how big the grid should be, as well as how many mines should be on it for the randomly generated boards."

How would you like the computer's ability to be measured?

"I would like to be able to measure the time the computer takes to solve a board, as that is how most players are judged, but having a measure of the difficulty of a board would be good too."

If the program were to have any extra features, what would they be?

> "Some way to track the computer's performance, so I can see how good it is compared to me."

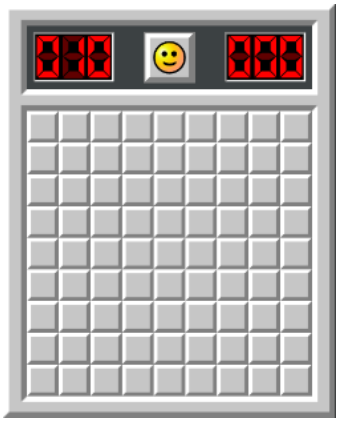
From this conversation I made a list of features the program should have:

- An implementation of the game which can be played both by the human and the computer.
- Some way to determine the computers ability to compare it against Chris or other users.
- The program should allow for the user to enter their own grids and have the computer solve them.
- Grid size and mine count should be customizable by the user.

With this in mind I started researching.

Rules

In order for some of the sections of this document to make sense, an understanding of the rules of Minesweeper, as well as some of the specialist terminology used is required.



Every game of Minesweeper starts with an empty grid of cells, two counters and a reset button as seen on the right. The counter on the left is the mine counter and tells you how many mines remain unflagged in the grid, and the counter on the right is the timer. The aim of the game is to clear the grid without clicking on a mine.

To start the game a cell must be clicked, in the standard version of Minesweeper you cannot uncover a mine with this first click. Each click will uncover one of two things, a number, or a mine. The number in the cell tells you how many mines are adjacent to it (including diagonals). Cells with no mines adjacent to them are blank and will uncover all adjacent cells automatically if clicked.

As well as left clicking to uncover cells, you can also right click to place flags. Placing a flag on a cell is a way of marking it as a mine, although flags can be placed on any unknown cells, reducing the mine counter by one. As the aim of the game is to uncover all non-mine cells, not to flag all mine cells, the flag isn't necessarily required to play the game and no-flag (NF) strategies can allow for more efficient although more difficult play.

The final action you can take is chording. If a cell containing a number is clicked then the game will count the number of flagged cells surrounding it. If the number of flagged cells around the clicked cell is equal to the number in the clicked cell then all non-flagged cells adjacent to it will be uncovered, whether they contain a mine or not. [1]

To determine the skill of a Minesweeper player there are a few metrics that are used. The time taken to solve a board is the most widely used determiner of skill, although the consistency with which a player can win is also an important factor.

Making the Game

My first task was to figure out how to create a fully functional game of Minesweeper. This section involves generating the board, dealing with scenarios which arise during the game and detecting when the game is finished.

Generating the Board

The first area I researched was board generation. The size of a board is usually displayed in the form *width x height / number of mines.* There are three primary difficulties of Minesweeper board:

- Beginner – 8x8/10 or 9x9/10 (depending on which version is being played)
- Intermediate – 16x16/40
- Expert 30x16/99

As well as these predetermined grids, I wanted the user to be able to generate boards with dimensions and mine counts of their choice. In order to keep the grid a reasonable size for a computer monitor I decided on a maximum of 50x50.

To generate a random Minesweeper grid takes two steps:

1. Mine Placement – A random number generator can be used for this, to generate cells repeatedly until the correct number of mines have been placed.
2. Filling the rest – Once the mines have been placed, every other cell should be filled in with a number indicating how many mines are adjacent to it.

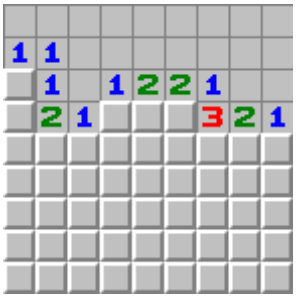
First Click

As stated before, the first click made in most versions of Minesweeper is guaranteed not to uncover a mine. This raises the issue of mine placements, as we don't know beforehand where the player will make their first click, we can't prevent a mine from being placed on that spot.

In order to circumvent this problem, an implementation of Minesweeper must, before revealing the first cell, check if that cell is a mine. If it turns out that the cell is a mine, then it should be moved, however by moving the mine, we now require the numbers on the board to be regenerated to account for the new location.

As you can see, the first click can require the program to partially recalculate the board, a process which will need to be managed carefully so no noticeable pause after the click is present.

Blank Cells



Blank cells also require some extra processing when clicked. In almost all versions of Minesweeper, clicking on a blank cell (one with no adjacent mines) will reveal all cells around it as well [2]. This is a simple enough feature to implement on its own, although gets more complex when we consider blank cells which also have blank cells adjacent to them.

It's very common for Minesweeper grids to have areas such as the one on the right, where there is a group of adjacent blank cells. The most efficient way of correctly uncovering these cells when one is clicked is a recursive flood-fill algorithm. An example of such an algorithm is shown below:

```
    For Each AdjacentCell
        If AdjacentCell is not Opened Then
            OpenCell
            If AdjacentCell is Blank Then OpenBlankCell(AdjacentCell)
        End If
    Next
```

End of the game

There are two ways in which a game of Minesweeper can end, either the player clicks on a mine and loses, or uncovers every cell which isn't a mine and wins.

The first scenario, clicking a mine, is trivial to detect. Information on what each cell contains will be stored, and a simple check after every click will determine whether the cell was a mine or not. Knowing whether the game has been won is slightly more complicated, however. The easiest, and not too computationally taxing, way to check is to iterate through every cell on the board and increment a counter for every cell which is open. If the counter is equal to (*width of board * height of board) – Number of mines on board* then all non-mine cells have been opened and the game is over.

Evaluating Success

Speed and Consistency

As stated before most Minesweeper players are judged by two metrics, speed and consistency. For a human, both are difficult to master, and increasing one usually decreases the other, for example when playing faster you are more likely to make mistakes. It is quite different, however, for a computer.

In Computer Science, time is a very valuable resource and algorithms which perform the same function faster are more desirable. This same logic is true of Minesweeper programs, although to a much lesser extent. The fastest time a human has ever beaten an expert game of Minesweeper is 29.461 seconds [1], for comparison a computer can solve an expert board in well under a second. For this reason, unlike humans, the speed of a Minesweeper solver is not usually used to determine its ability, instead its consistency is the main factor.

Consistency is more difficult to measure, although most use their win percentage (the ratio of games won against games played). From some browsing of forums such as Reddit[2], I have found that somewhere between 25 – 35% seems to be considered the level of a skilled player though a lot of that data was collected from small sample sizes of hundreds of games due to the time-consuming nature of humans playing Minesweeper (games can take 5-10 mins each when going for consistency). For this reason, I decided to aim for my solver to have a win percentage within this range, and for it to hold across a much larger sample size. I deemed the speed of my solver to be less important, although it should be quick enough to not inconvenience the user.

Difficulty

On other measure of skill, although used less often, is difficult. The difficulty of a Minesweeper board is judged by its 3BV (Bechtel's Board Benchmark Value). The 3BV of a board is the minimum number of clicks required to solve it (not considering flagging or chording). 3BV is calculated before the game by using a very similar flood-fill algorithm to that used for blank cells. A description of the algorithm from user Michael Madsen on Stack Exchange can be found below [3]:

1. First the board should be converted into an array containing the values in each cell on the board (this technique is used extensively in the solver)

```
0000002M
0000013M
110113M3
M101M3M2
11011222
0000001M
00122222
001MM2M1

Clicks = 0
```

2. The algorithm "marks" the cells as it goes, so to start off with every cell is unmarked (.)

```
........
........
........
........
........
```

```
........
........
........

Clicks = 0
```

3. We now mark (*) the top left cell to start with, indicating one click

```
*.......
........
........
........
........
........
........
........

Clicks = 1
```

4. In this case the top left cell was a “0”, meaning if it was clicked all adjacent cells would open as well without any more clicks being used. For this reason, each area of “0”s and the numbers immediately adjacent to them can be marked as only needing one click.

```
*******.
*******.
******..
.***....
*******.
*******.
*******.
***.....

Clicks = 1
```

5. Each non-mine cell which isn’t adjacent to a “0” will take one click to open, in this case there are 7 such cells (#). We can increment the number of clicks for each one.

```
*******.
*******.
******.#
.***.#.#
*******#
*******.
*******#
***..#.#

Clicks = 8
```

We now have the minimum number of clicks required to beat this board, 8. A 3BV of 8 is very low, meaning this board would be easy to solve. 3BV can vary a lot, even on boards that have the same size and number of mines, depending on where the mines are placed. The 3BV of these boards follow a normal distribution [4], although most expert boards lie in the mid to high 100’s. (interestingly there are only 12 games in the top 100 fastest times with a 3BV above 150 [1])

I don’t want to pay too much attention to difficulty when testing my solver, as it is extremely luck based to solve very high difficulty boards so isn’t a useful measure of success.

Tracking Statistics

In order to properly evaluate the skill of my solver, I will need to include a way to track its statistics and store them in a file. Each board the computer plays will be tracked, and the following data will be stored for each to allow me to determine the computers skill level and evaluate my success:

- **Board dimensions and mine locations** – allows me to recreate boards to see where the computer went wrong, as well as judging how difficult the board was.
- **Time taken to solve** - allows me to evaluate the speed of the computer when solving (although this metric isn't overly important).
- **3BV** – another way for me to determine the difficulty of boards the computer solves.
- **Result** – whether the computer won or lost each board allows me to compute its overall win percentage.
- **"Hardest" algorithm used** – for use in the training function explained later.

Training and Custom Boards

The best way to use custom boards that I have found, which has the added benefit of aiding the user in improving at Minesweeper is an "analysis board" such as the one in an implementation of the game made by Drew Roos (AKA MrGris)[6] which allows the user to create their own Minesweeper board and have the computer inform them of any safe cells and mines, and the best guess to make if it gets stuck. This feature is identical to the user generated requested by Chris.

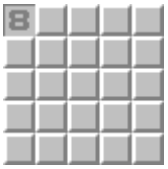
Making a Custom Board

In order to have a custom/analysis board feature in my program, there are several issues which need to be addressed.

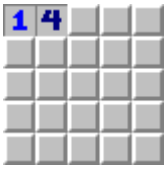
The first issue is size of board, if the user is able to generate their own boards, then I need to make sure they can only generate ones which don't violate the following rules:

- All boards must be greater than 0 in both width and height.
- No board can have a mine count greater than or equal to the number of cells on the board
- Boards should have a maximum size of 50x50 (for same reason as before)

Another issue that arises with custom boards is that of validity. If the user is allowed to enter any values they like into cells, they can create any board they want. The board can only be solved by the computer, however, if it is a valid Minesweeper board so the validity of a board will need to be checked before any attempt is made at solving it. Two examples of invalid boards are shown below:



This Board is very obviously invalid, an 8 cannot border only three cells.



This board is also invalid, although in a more subtle way. The 4 tells us that all 4 adjacent cells must be mines. Although the 1 tells us that only one of the two cells adjacent to it can be a mine. This a contradiction, therefore the board cannot be valid.

The user should be able to choose a size of board (within reason) and input any values they like into the cells. The computer should then solve the board (provided it is valid) and display the results, including any safe cells, mines and guesses. Other features that I decided to include for ease of use include:

- The option to save an image of the board to the user's computer.
- The option to import an image of a Minesweeper board (only one's generated by my program to avoid complications) and edit/solve that board.
- Shading over cells to indicate their status.
- The option to reset a board to a blank grid.

All images of Minesweeper boards used in this document have been generated and saved using my program.

Finalising Objectives

Before finalising my objectives, I consulted with my end user, Chris, to confirm he was happy with my planned program.

Is a maximum grid size of 50x50 acceptable?

> "I don't usually play big boards, so that's fine, as long as I can generate any board up to that size."

My research has indicated that time isn't a good measure of the computer's ability, so I have decided to judge it based on win rate. Is there any other metrics you want measured?

> "I would ideally like to still be able to time my own games, as well as the computers, even if win rate is a better measure of skill. I would like to be able to see, for each game, whether the computer won the game or not as well."

How would you want the information that is stored about the computers ability to be laid out?

"If possible, a real time tracker as the computer plays would be useful, so I can see how it is performing as it plays. As well as this, some form of table which contains information about each game the computer has played."

Would you be interested in being able to return to past boards the computer has played and be able to play them yourself?

> "That would be a useful feature to have, if possible, could you make it such that I can return to a game I have played before as well, in case I make a mistake and want to retry a board."

From this conversation, as well as the research I carried out, I decided on a list of objectives which my program will aim to fulfil, and will be tested thoroughly at the end:

Required:

1. The program should be able to generate any size of board, up to and including 50x50.
2. The program should allow any number of mines on this board, provided it is less than the number of cells on the board.
3. When either a human or computer clicks the board for the first time in a game, it should always be safe.
4. When blank cell is opened, all adjacent cells should also be opened, this should be recursive for any blank cells opened.
5. When a numbered cell is clicked, if there are the correct number of flags adjacent to it, then all non-mine cells adjacent should be opened (Chording).
6. When all non-mine cells have been opened, the user should be informed that the game has finished, and they have won.
7. When a mine cell is opened, the user should be informed that the game has finished, and they have lost.
8. When the reset button is pressed, a new randomly generated board with the same dimensions and same number of mines should be created for the user to play.
9. The speed of the Minesweeper solver should be such that there are no breaks for computation longer than 5 seconds that inconvenience the user of the program.
10. The Minesweeper solver should have a win rate of between 25 and 30% when measured over a sample of 10,000 expert boards.
11. After every game, the stats for that game (including difficulty, result, time etc) should be written to a human readable stats file contained within the programs resource folder.
12. A visual display of the stats for every game played by both human and computer using the program should be available for browsing.
13. These stats should be filterable and sortable by Difficulty, Result, Time and 3BV.
14. A live tracker of the results of the solver, including a running tally of games and the current win rate, as it is playing the game should be available for the user when the computer is playing the game.
15. The user should be able to create any Minesweeper board up to and including a 50x50 grid, either by manually inputting values or using an image.
16. The computer should detect if a user generated grid is valid, and if so, correctly determine the locations of any mines, safe cells, or best guesses and visually display that data to the user by shading the cells on the board with different colours.
17. The user should have the ability to save an image of any Minesweeper board they have created directly to their computer using the computers built in file explorer.

Optional:

1. The user should be able to choose any game from stats, and have that board recreated for playing again.
2. Give the user the option to choose different difficulties of boards and generate a board of that difficulty which can be played by either human or computer.

Design

The following section outlines the algorithms and data structures used within the program as well as their purpose.

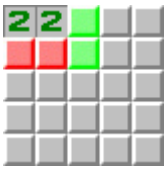
Playing the Game

Before explaining the algorithms used by the computer to play the game, a description of how humans approach Minesweeper is needed.

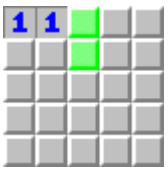
The Human Approach

For a game designed to be played by humans, Minesweeper lends itself extremely well to a heavily computational approach to playing, the opposite of what the human brain excels at. Despite this, we have found ways to play Minesweeper without having to rely on using large amounts of calculation, namely pattern recognition.

Human players almost exclusively play the game by memorizing patterns and their solutions and applying them to solve a board. The simplest patterns are as trivial as recognizing that if a number is immediately adjacent to the same number of cells, as the leftmost 2 is here, then they must all be mines, and if the number is immediately adjacent to the same number of mines, as the rightmost 2 is here, then the rest of the adjacent cells can be opened.



Another simple pattern to recognise is the 1-1 pattern. The leftmost 1 indicates that there must be a mine in one of its two adjacent cells. The rightmost 1 is also adjacent to these two cells, which already must contain a mine, therefore the other cells it is adjacent to can be opened safely.

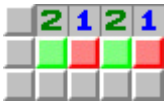


There are countless patterns to be recognised, although most can be boiled down to a mix of simple patterns such as the 1-1 or the 1-2 [5] using a technique called reduction. Using reduction is a major

difference between new Minesweeper players and experienced ones. A simple example of reduction is this board:

As some of the mine locations are already known thanks to the top row of numbers, we can "reduce" some of the other cells to help us solve this pattern. To reduce a cell is to decrease the number inside it by the number of mines it is adjacent to. In this example the numbers on the third row can all be reduced using the mines above them, leaving us with this board (ignoring the top two rows as they now have no relevance to us):

This is now an easily solvable board, the 1-2-1 pattern (which is actually just two 1-2 patterns next to each other) is very common and is solved like this:



Finally, taking this information back to the original pattern, we can now deduce the safe cells on the unreduced board by copying this result over:

An experienced Minesweeper player will immediately notice these patterns and reductions, and know the correct configuration to solve them. This is likely a result of the way humans think, where pattern recognition is a valuable and a regularly used skill in everyday life. This means human players play Minesweeper based largely on intuition rather than pure logic, as an AI might do if it were to attempt the game.

A computer on the other hand is much better at implementing logic to arrive at its conclusion and therefore isn't suited to a pattern-based strategy. This allows them to excel in situations where humans would struggle, such as when a guess is required, and a time-consuming calculation is the only way to determine the best possible move.

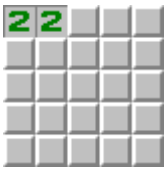
An overview of the common basic patterns, as well as explanations for more complicated strategies such as mine counting and combinations (which also are covered later in this document) can be found at https://minesweeper.online/help/patterns

The Computational Approach

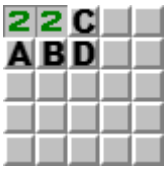
When a computer plays Minesweeper it, for the most part, uses the same techniques as a human would, but modelled in a more rigorous mathematical fashion. I have broken this down into three sections, simple solving (trivial patterns), non-trivial constraint satisfaction (reduction) and brute-force.

Simple Solving

Simple solving is a logical representation of how a human would recognise and solve a trivial pattern. it involves forming and satisfying constraints, here is an example using the same board as for the corresponding human approach:



Our first step is to label all the cells adjacent to the two numbers (this would usually be done with their coordinates on the board).



We can now generate equations using these cells. Each equation is formed by taking one of the cells containing a number and equating that number to the letters adjacent to it. This forms an equation, known as a "constraint" which tells us which cells could contain mines. The constraints for this pattern are:

$$A + B = 2 \text{ (from leftmost 2)}$$

$$A + B + C + D = 2 \text{ (from rightmost 2)}$$

The right hand side of the constraint tells us how many mines there are in the cells on the left hand side. To solve these constraints we assign values to the cells, a cell containing a mine has value *1* and a cell not containing a mine has value *0*. Using this information we can see that to satisfy the first constraint, both values must be *1*, and therefore both *A* and *B* must be mines. We can substitute our values for *A* and *B* into our second constraint:

$A + B = 2 \rightarrow A = 1, B = 1$

$1 + 1 + C + D = 2$

From here it is obvious that both *C* and *D* must have a value of *0* to satisfy the constraint, and therefore be safe. Transferring this information onto the board we can generate the solution to this pattern, and see that it is identical to the one found using the human approach:

Comparing this to the logic applied by humans we can see that, in essence, these constraints are saying the same thing

Human	Computer
If the number of cells adjacent to a number is equal to the number then they are all mines.	$A + B = 2 \rightarrow A = 1, B = 1$
If the number of mines adjacent to a number is equal to the number, all other cells adjacent to it are safe.	$1 + 1 + C + D = 2 \rightarrow C = 0, D = 0$

Here is an algorithm in pseudocode for the simple solver:

```
For Each Cell on the Board
            If NumofAdjacents = NumberinCell Then
                  Adjacents = Mine
            End If
            If NumofMines = NumberinCell Then
                  NonMineAdjacents = Safe
            End If
      Next
```

A simple solver algorithm is able to solve the majority of Minesweeper positions, especially on lower difficulty boards. In order for the computer to solve patterns such as 1-1 and 1-2, a slightly more complicated algorithm is required.

Non–Trivial Constraint Solving

This new algorithm also makes use of constraints on the form of equations to solve positions. The difference between trivial constraint solving (simple solver) and non-trivial constraint solving is the use of substitution, which allows this algorithm to "recognise" and find solutions to the more complex patterns found in Minesweeper. This algorithm also makes use of reduction. here is how it would solve the reduction example used earlier:

As before, the first step is to label the unknown cells adjacent to the numbers:

Before generating our constraints, we need to reduce this pattern. this is done by checking each number for any known mines adjacent to them, and decrementing them for each:

We can now generate constraints:

$$A + B + C + D = 2$$

$$C + D + E = 1$$

$$D + E + F = 2$$

$$E + F = 1$$

As you can see, none of these equations can be solved using the "trivial" method of the simple solver as the number on the right-hand side is less than than the number of variables on the left, although they can be solved using substitution. Considering the last two constraints, we can deduce the value of *D* by substituting one into the other:

$$D+E+F=2, E+F=1 \rightarrow D+1=2 \rightarrow D=1$$

With the knowledge that $D = 1$ we can now find the value of *C* and *E*:

$$C+1+E=1 \rightarrow C=0, E=0$$

And then finally, using $D = 1$, $C = 0$, $E = 0$ we can find the value of *F*:

$$0+F=1 \rightarrow F=1$$

We can't find the values of *A* or *B* yet as our equation still has multiple unknown variables, and therefore multiple ways to satisfy the constraint, although we do know that only one of them can be a mine:

$$A+B+0+1=2 \rightarrow A+B=1 \rightarrow A=1, B=0 \; OR \; A=0, B=1$$

Putting these values back into our reduced pattern we get this result:

And then back into the original pattern to give the same solution that we found using the solution to the 1-2-1 pattern:

The non-trivial constraint solver allows the computer to be more adept at Minesweeper than the simple solver, although is essentially just an extension of it, and doesn't make for a computer which can outplay a skilled human.

Brute Force

The final strategy utilised by Minesweeper solvers is a brute force algorithm. This algorithm allows computers to play the game at a higher level than any human (if programmed efficiently) as it involves relatively complex and very time-consuming calculations. The computers ability to perform these calculations so quickly can allow for insights that wouldn't be possible to spot by a human using pattern recognition and, most importantly, it allows for the computer to play as optimally as possible if it can't find a definite safe cell. Before discussing the workings of the algorithm, the problem of NP-Completeness and its relevance to Minesweeper needs to be covered.

NP-Completeness

NP-Completeness (Nondeterministic Polynomial-time Completeness) is a fairly advanced concept in Computer Science, any problem which is "NP-Complete" can be solved in a polynomial time complexity by a Nondeterministic Turing Machine. It applies to many other fields as well and is well beyond the scope of this project. More information on what it means for a problem to be "NP-Complete" can be found here[7].

NP-Completeness directly relates to a very famous problem, which is usually presented with the statement **P=NP**. This statement is saying that any problem which can be solved in a polynomial time complexity by a Nondeterministic Turing Machine (NDTM) can also be solved in a polynomial time complexity by a Deterministic Turing Machine (DTM). NDTM's are (as of now) a purely theoretical concept and cannot exist in the physical world, whereas DTM's are, effectively, computers and any program that can be solved with a DTM can also be solved with a computer (and vice versa). The statement of P=NP has not been verified to be true (or untrue) yet, meaning currently problems which can be solved in polynomial time with an NDTM can't necessarily be solved in polynomial time by a DTM. More information on the concept of NDTM's and DTM's can be found here[8].

In 2000, it was proved that Minesweeper is an NP-Complete problem[9]. The practical implications of this proof is that it is impossible to find a solution for all possible Minesweeper positions within a polynomial-time complexity, although a solution can always be verified within this time complexity. This means that a brute force solver for the game will never be perfect, and some restrictions will have to be placed onto this algorithm to keep it's exponential-time complexity from inconveniencing the user.

Interestingly, if anyone can prove that Minesweeper (or any other NP-Complete problem for that matter) can be solved in polynomial time using a DTM then it would prove the same result for all other NP-Complete problems as each of these problems can be reduced to all of the others. If anyone ever proves the result P=NP (or P≠NP) then they are eligible to claim a million-dollar prize from the Clay Mathematics Institute as this is one of their seven "Millennium Problems"[10].

The Algorithm

The brute force algorithm makes use of the constraints generated by the previous two methods , and is used when no solution can be found to these constraints by either of them. At this point, the computer has to resort to checking every possible solution to the constraints and analysing them for consistencies. Here is an example where the brute force solver can find a solution to an otherwise unsolvable position:

In this position, some mines have been identified and marked already using the other algorithms. We first need to reduce the pattern and label the cells we are interested in:

And now we can generate our constraints, this will already be done by previous algorithms in the program and passed onto the brute force solver :

$A + B = 1$ *(repeat doesn't have any effect)*

$B + C + D + G + H = 1$

$I + J + K = 2$

$G + J + K + L = 1$

$E + F = 1$

Before any solutions are generated, we must first group some of these constraints. As the values in each constraint are not independent of each other, all constraints which share at least one variable should be considered together. The groups are as follows:

Group 1:

$A + B = 1$

$B + C + D + G + H = 1$

$I + J + K = 2$

$G + J + K + L = 1$

Group 2:

$E + F = 1$

Now we have groups of constraints, we can start to generate solutions. If we want to find safe cells or mines, then we have to generate all possible solutions to the equations, which is done using a backtracking algorithm and stacks. As *Group 2* only contains one equation, the solutions to it are trivial to find, so we can list them here (the program would still run the backtracking on this group to find the solutions):

Solutions to Group 2:

- $E = 1, F = 0$
- $E = 0, F = 1$

Group 2's solutions do not provide us with any useful information about the current position so we need to find *Group 1's* solutions which are less obvious. The first step to finding these solutions is to create 2 stacks, one for storing the variables we are considering (*VariableStack)*, and the other to store the variables we need to return back to later (*ReturnToStack)*. A list of all the variables along with their current value, which is 0 for each at the start, contained in *Group 1* is also created (*VariableList)*:

VariableStack	*ReturnToStack*

VariableList:

Variable	*Value*
A	*0*
B	*0*
C	*0*
D	*0*
G	*0*
H	*0*
I	*0*
J	*0*
K	*0*
L	*0*

We are now ready to start generating solutions. To do this we start with the first variable in the list, and swap its value from *0* to *1* or vice versa. Then we refer back to our equations and check that there are no contradictions, meaning that the left-hand side of all equations containing this variable are less than or equal to the right-hand side of those equations:

$$A = 1 \rightarrow 1 + B = 1 \therefore \text{No Contradiction}$$

As there is no contradiction, we can safely push this variable to *VariableStack*. As the value of *A* has been set to *1*, we must also push it to *ReturnToStack*, indicating that at some point we must return to this variable and consider the constraints when its value is *0*:

VariableStack	*ReturnToStack*
A (1)	*A*

This process is now repeated for the next variable, *B*. As *A = 1*, the first of our constraints now has a contradiction, meaning that *B* must be set to 0 in this solution, and can be pushed to *VariableStack*, but does not need to be pushed to *ReturnToStack*:

$$B = 1 \rightarrow 1 + 1 = 1 \therefore \text{Contradiction} \therefore B = 0$$

VariableStack	*ReturnToStack*
B (0)	
A (1)	*A*

After each variable has been considered in this way (not shown as process is the same for each), and the number of variables in *VariableStack* is equal to the number of variables in *VariableList*, we have finished, and can work out if we have generated a solution to our set of equations by substituting our values in and checking if all of the constraints are satisfied:

VariableStack	*ReturnToStack*
L (0)	
K (0)	
J (1)	
I (1)	
H (0)	
G (0)	
D (0)	*J*
C (1)	*I*
B (0)	*C*
A (1)	*A*

VariableList:

Variable	*Value*
A	*1*
B	*0*
C	*1*
D	*0*
G	*0*
H	*0*
I	*1*
J	*1*
K	*0*
L	*0*

$A\,(1) + B(0) = 1 \therefore Satisfied$

$B\,(0) + C\,(1) + D(0) + G\,(0) + H(0) = 1 \therefore Satisfied$

$I\,(1) + J(1) + K(0) = 2 \therefore Satisfied$

$G(0) + J(1) + K(0) + L(0) = 1 \therefore Satisfied$

As all four of our constraints are satisfied, we know that this set of values is a possible solution to the position, and we create a list of solutions and add to it the current values in *VariableList*:

SolutionList

Solution	*VariableListValues*
1	*1, 0, 1, 0, 0, 0, 1, 1, 0, 0*

We now need to find a new solution. This is done by comparing the variable at the top of *VariableStack* with the variable at the top of *ReturnToStack*. If they are not the same, then *VariableStack* is popped, and then the top of it is compared again until they are the same, and both are popped:

VariableStack	*ReturnToStack*
I (1)	
H (0)	
G (0)	
D (0)	
C (1)	*I*
B (0)	*C*
A (1)	*A*

In order to generate a new, unique solution we need to repopulate *VariableStack* with the variables that were popped from it. In order to do this, we just repeat the process of adding variables to the

stack that was carried out before starting from *J* , which now has a value of *1*, and so is first changed to a *0*. The rest of the variables are calculated from there to rebuild the stacks as follows:

VariableStack	*ReturnToStack*
L (0)	
K (1)	
J (0)	
I (1)	
H (0)	
G (0)	
D (0)	*K*
C (1)	*I*
B (0)	*C*
A (1)	*A*

This new solution is checked to verify that it satisfies the constraints, then added to *SolutionList*. This entire process is repeated, until there are no variables remaining in *ReturnToStack*, indicating that all possible solutions have been found. After this, *SolutionList* should be as follows:

SolutionList

Solution	*VariableListValues*
1	*1, 0, 1, 0, 0, 0, 1, 1, 0, 0*
2	*1, 0, 1, 0, 0, 0, 1, 0, 1, 0*
3	*1, 0, 0, 1, 0, 0, 1, 1, 0, 0*
4	*1, 0, 0, 1, 0, 0, 1, 0, 1, 0*
5	*1, 0, 0, 0, 0, 1, 1, 1, 0, 0*
6	*1, 0, 0, 0, 0, 1, 1, 0, 1, 0*
7	*0, 1, 0, 0, 0, 0, 1, 1, 0, 0*
8	*0, 1, 0, 0, 0, 0, 1, 0, 1, 0*

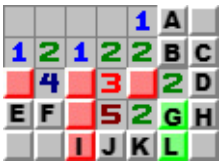
As this is an exhaustive list of solutions, any variable which is *1* in every solution must be a mine, and any variable which is *0* in every solution must be safe:

G = 0 in all solutions ∴ G is safe

I = 1 in all solutions ∴ I is a mine

L = 0 in all solutions ∴ L is safe

Finally, we can use this information to solve the original position as shown:



This algorithm isn't usually used for finding safe cells/mines, as the previous algorithms are usually sufficient to find any guaranteed safe cells, but is instead used to inform the final algorithm used by the computer to play Minesweeper, the guessing algorithm.

Guessing

When all three previous algorithms have been carried out, and no safe cells have been identified, the computer has no choice but to guess at the best cell to open. This can be done in two ways, local guessing and global guessing. The example used to demonstrate the difference between the two methods has been taken from Sean Barret's Minesweeper: Advanced Tactics[11].

Local Guessing

Local guessing considers groups of cells independently from each other, which makes it much more time and space efficient, but also prone to inaccuracy. Here is the board being used for this example:

This board has almost been completed, although there are now no cells in this board which we can say, with certainty, are safe to open. To continue we must make a guess, first of all though we need to identify our groups:

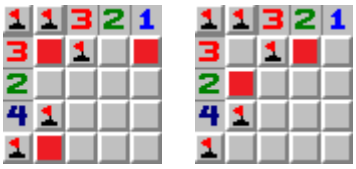
The next step is to generate the solutions to each group using our brute force algorithm, Group 4 has no solutions, as we currently have no information about the cells within it so it can be ignored. The solutions for the other groups are shown below:

Red indicates that a mine is in this cell.

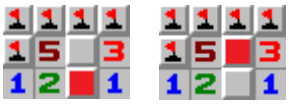
Group 1

Note that the yellow cells are independent of the rest of the group, and therefore have no bearing on the location of any other mines.

Group 2



Group 3



When guessing locally, we would now consider each group, one at a time, and determine the probability that each cell in that group is a mine. To determine the overall probability that a cell is a mine, you divide the number of solutions in which that cell is a mine by the total number of solutions:

Group 1

A and C are independent of the rest of the pattern, there is one mine which could be in either cell, so the probability of it being in each is 50%.

B and G are mines in 2 of the 5 solutions, the probability of either being a mine is therefore 2/5 or 40%.

D, E and H are mines in 3 of the 5 solutions, the probability of each being a mine is therefore 3/5 or 60%.

F is a mine in 4 of the 5 solutions, the probability of it being a mine is therefore 4/5 or 80%

A – 50% | B – 40% | C – 50% | D – 60% | E – 60% | F – 80% | G – 40% | H – 60%

Group 2

All of the cells in Group 2 are mines in only one of the two solutions, therefore the probability of each being a mine is 50%

I – 50% | J – 50% | K – 50% | L – 50% | M – 50%

Group 3

Both cells in Group 3 are a mine in one of the two solutions therefore, similarly to A and C in Group 1, each has a 50% chance of being a mine.

N – 50% | O – 50%

If we were to decide which move to make next based on these local probabilities, we would be best off choosing either *B* or *G* from Group 1 as they have the lowest probability of being a mine.

This method of guessing can be quite naïve however, as the groups are less independent of each other than it first may seem. None of the examples given to explain the algorithms so far have made use of the mine counter, a vital piece of information supplied to the user by the game. Global guessing uses these local probabilities, along with the mine counter, to generate more accurate probabilities for the cells.

Global Guessing

Global guessing differs from local guessing in that it does not assume that each group is independent, more mines present in one group would mean less mines available for the other groups. It also uses the fact that mines can be present in cells which we have no information about,

such as Group 4 in our example. Before calculating global probabilities, we need to organise our solutions by number of mines:

Group 1

A – 6 Mines

B - 5 Mines

C - 5 Mines

D - 5 Mines

E - 4 Mines

Group 2

F – 3 Mines

G – 2 Mines

Group 3

H – 1 Mine

I – 1 Mine

Group 4

Unknown number of mines, but must be between 0 and 12 as there are 12 cells in Group 4.

From this we can see that the maximum number of mines that could be left on this board would be *A(6) + F(3) + H/I(1) + 12 = 22* and the minimum number would be *E(4) + G(2) + H/I(1) + 0 = 7*, although it is extremely unlikely all cells in Group 4 are mines and also very unlikely that no cells in Group 4 are mines.

Assuming the original board was an expert board, meaning it contains 99 mine in total, by counting the number of flags we have placed already (or using the mine counter in the top left of standard versions of Minesweeper), we can find out how many must be left in unopened cells. This value comes out to **8** for this particular game.

We can now generate a list, which contains every possible combination of these solutions and how many mines would be contained in them. As Group 3 has only two solutions which both have the same amount mines, they can be treated as identical as the outcome of choosing one over the other has no bearing over the rest of the board. For this reason, we can take 1 from the mine count, and continue the algorithm without considering Group 3. Here is a list of all possible combinations of solutions along with the number of mines in each:

Total mines on board – 8 (discounting one for Group 3)

Solutions	*Mines*
A, F	*9*
B, F	*8*
C, F	*8*
D, F	*8*
E, F	*7*
A, G	*8*
B. G	*7*
C, G	*7*
D, G	*7*
E, G	*6*

Note that Group 4 hasn't been included in the solutions, as we had no information to deduce which of the cells are more likely to be mines. Although we cannot determine the location of the mines in Group 4, we can now say for each combination of solutions, how many mines would be left. All of these remaining mines must be in Group 4.

As we have no other information about their location, we can assume that it is equally likely for these mines to be in any of the cells in Group 4. Using this assumption, we can work out how many unique ways the remaining mines can be arranged in the group by using the nCr function. There are 12 mines in Group 4 so for each combination of solutions, the number of permutations of the remaining mines can be found using **12C(NumMinesRemaining – NumMinesInSolution)**. The combination of *A* and *F* has 9 mines, meaning it is impossible on this board and can be removed from consideration. Adding this information to our table gives us this:

Total mines on board – 8 (discounting one for Group 3)

Solutions	***Mines***	***Permutations***
B, F	*8*	*12C0 = 1*
C, F	*8*	*12C0 = 1*
D, F	*8*	*12C0 = 1*
E, F	*7*	*12C1 = 12*
A, G	*8*	*12C0 = 1*
B, G	*7*	*12C1 = 12*
C, G	*7*	*12C1 = 12*
D, G	*7*	*12C1 = 12*
E, G	*6*	*12C2 = 66*

Total permutations = 1 + 1 + 1 + 12 + 1 + 12 + 12 + 12 + 66 = 118

We now need to return to our individual solutions. We can calculate, of the 118 possible permutations, how many of them include each solution by summing up the permutations of

combinations they appear in. For example, *B* appears in two combinations (*B, F* and *B, G*) which account for 13 of the 118 total permutations. Here is the sum for each solution:

Solution	*Permutations*
A	*1*
B	*13*
C	*13*
D	*13*
E	*78*
F	*15*
G	*103*

The final step is to consider the cells in each solution. To find their global probabilities, we find all the solutions in which the cell is a mine, and sum them up. We then divide this by the total number of solutions (118) to find a probability that the cell is a mine. This is effectively asking *Out of every possible way to arrange the remaining mines, in how many is this cell a mine?* Here are the number of permutations for each cell and their respective percentage probabilities of being a mine:

<table>
<tr><th>Group</th><th>Permutations</th><th>Probabilities</th></tr>
<tr><td>Group 1</td><td> 
</td><td> 
</td></tr>
<tr><td>Group 2</td><td> 
</td><td> 
</td></tr>
</table>

We can now deduce which cell is least likely to have a mine contained within it. In this case there are two cells in Group 1 with a 12% chance of being a mine, making them the safest option for the next move.

Random Cells

To add on to the global guessing algorithm, we can also find the probabilities that one of the cells in Group 4 is a mine. As we have previously assigned them all equal probability due to our lack of information, we only need to perform this process once.

To do this, we look back to our combinations of solutions, and how many mines each one had. As any remaining mines in these combinations would have to be in one of the 12 cells of Group 4, we can assign a probability of *RemainingMines/12* for the chance of each containing a mine.

Finding an overall probability is finding an average of these individual probabilities, which can be done in a variety of ways although I chose to use a simple mean calculation which is accurate enough for almost all cases. As each solution has a number of permutations, the data can be thought of as cumulative, and the frequency (number of permutations) should be taken into account when finding a mean. The table below shows each individual probability for Group 4, then an average overall probability, calculated by dividing the sum of individual probabilities by the total number of permutations:

Solutions	*Permutations*	*Probability*
B, F	*12C0 = 1*	*0/12 = 0*
C, F	*12C0 = 1*	*0/12 = 0*
D, F	*12C0 = 1*	*0/12 = 0*
E, F	*12C1 = 12*	*1/12*
A, G	*12C0 = 1*	*0/12 = 0*
B, G	*12C1 = 12*	*1/12*
C, G	*12C1 = 12*	*1/12*
D, G	*12C1 = 12*	*1/12*
E, G	*12C2 = 66*	*2/12 = 1/6*

Overall Probability = (12*1/12 + 12*1/12 + 12*1/12 + 12*1/12 + 66*1/6)/118 = 15/118 = 0.13

In this situation clicking a cell in Group 4 would leave a 0.13 chance of losing, very slightly worse than the two cells in Group 1, therefore we have calculated, and fully justified our best next move, and can now play it and hope some useful information is revealed.

Corners and Information

If our calculations decided that opening a cell from Group 4 was the best option, we would then have to make a decision about which one. As we have no information about any, you'd expect each to be equally as good, although this is not the case.

Although it is not used in my program, several solvers exist for Minesweeper which use an extra parameter when deciding on a next move, information. Some cells on the board are more likely to reveal useful information (that would allow you to more safely solve the rest of the board), and therefore they should be valued higher.

It can be shown that, when presented with the option of clicking a random cell with no other information, there is a "best" option, a corner. The most useful cell to a player in Minesweeper is a blank one, as it reveals the most information, therefore the best move would be one that is most likely to reveal a blank cell. The likelihood of opening a blank cell can be roughly estimated using the mine density of the board (number of mines divided by number of cells) and the number of cells adjacent. If none of the adjacent cells are mines, then the cell will be blank:

For an Expert Board (Mine density = 0.21):

Corner Cell (3 adjacent cells) : $P(Blank) = (1-0.21)^3 = 0.50$

Edge Cell (5 adjacent cells) : $P(Blank) = (1-0.21)^5 = 0.32$

Center Cell (8 adjacent cells) : $P(Blank) = (1-0.21)^8 = 0.16$

(Note that this assumes the cell you opened wasn't a mine, as if it was then it wouldn't matter what the probabilities are because you've lost)

This tells us that opening a corner is more than 3x more likely to result in a blank cell than opening in the center, therefore we should always strive to click a cell in the corner when possible.

This strategy also applies directly to the start of the game, where no information is known, so starting in a corner is the best option.[12]

Problems with Brute Force

A well-known problem, not only with this algorithm, but with all brute force algorithms is their scalability. Both the initial solving and the global guessing algorithm require steps that have an exponential (or worse) time complexity. The easiest way to get around this is to limit the number of items that can be considered at a time to make sure the computation time remains bearable. I do this in two ways in my program:

1. When combining constraints using common variables, the maximum number of variables that can be combined into one set of equations is 20. From testing I realised that any more can result in the backtracking algorithm used to find an exhaustive list of solutions taking several minutes to complete, which would render my program almost unusable.
2. When attempting to find a guess, my program defaults to using global guessing as it is more accurate, however, if there are too many combinations of solutions then this algorithm can also take a very long time to complete. For this reason, if the total number of combinations

is more than 500000 then the program will use the local guessing algorithm instead, which is less accurate but much more computationally efficient.

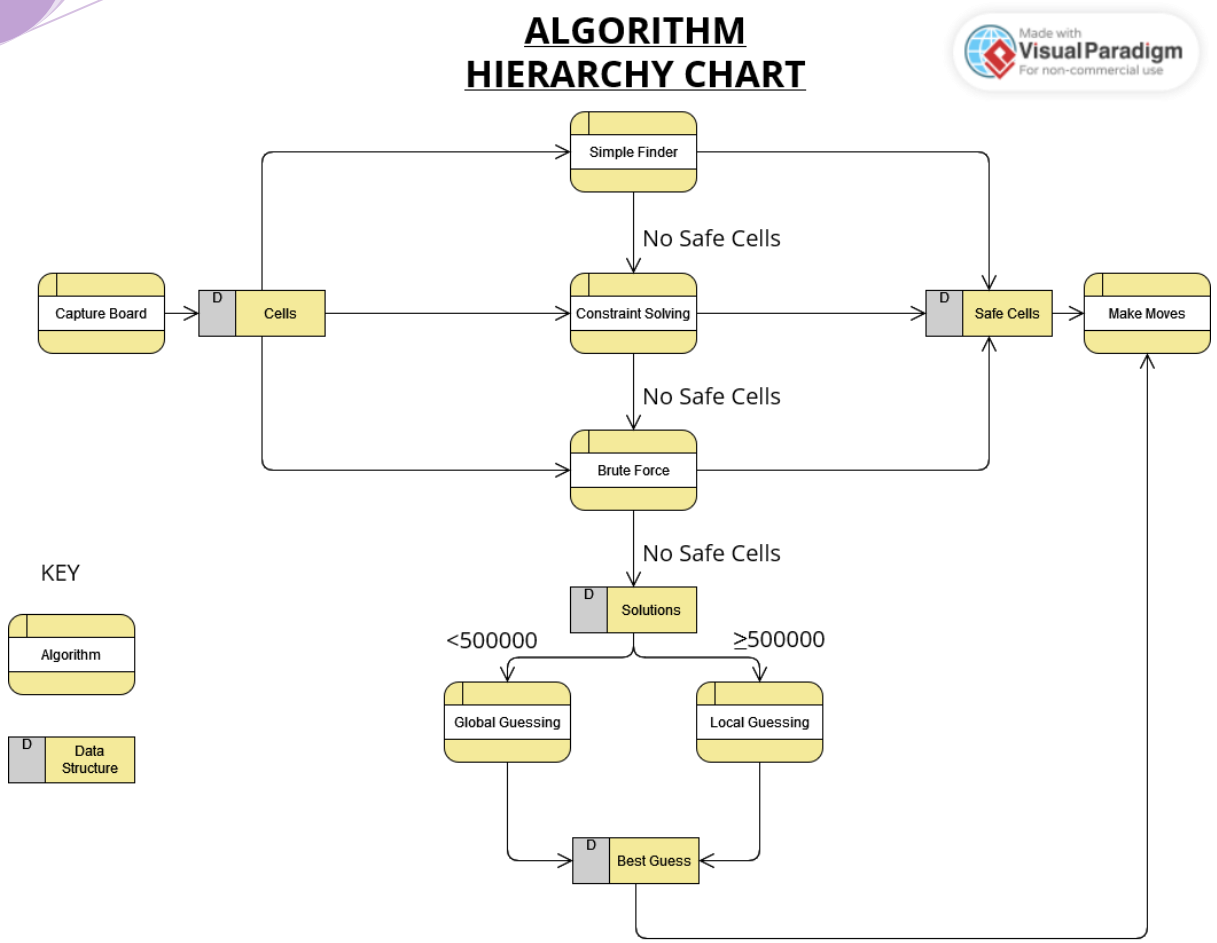
Structure and Hierarchy

In order for my solver to function as intended, and do so in the most efficient way possible, my algorithms are structured in a hierarchy, which dictates in which order they are run.

Once the board has been captured and stored, the first algorithm to be run is the Simple Finder. This will iterate through each cell and store any it deems to be safe. If, after this has been run, there are no safe cells, then the Constraint Solver will be run next. This will again determine any safe cells and store them in a data structure. If there are still no cells which can be opened, then the computer will resort to the brute force backtracking algorithm. This generates an exhaustive list of solutions and analyses them for patterns indicating safe cells. If there are still none to be found then the computer has to guess. The algorithm used for guessing depends on the number of combinations of solutions, if this is less than 500000 then global guessing is used, otherwise local guessing is used and the resulting "best guess" is stored. Cells are now opened and the process is repeated with the new information.

Note that if a safe cell is found by an algorithm, that algorithm will finish, but the next one down in the hierarchy will not be run, and instead the known safe cells will be opened and the process restarted.

Here is a chart detailing this process:



Class Overview

This program uses a variety of different classes in order to play the game, below is an overview of each, including their attributes, methods, and purpose:

Game

Attributes

Name	Type	Purpose
Cells	*Cell*(,)	The most important data structure used in the program. Each location in the array contains a *Cell*, which holds the information about the cell at that index on the grid.
SafeCells	List(Of *Cell*)	Stores a list which is appended to with any safe cells that are found, this list is regularly iterated through and emptied

		when the cells are opened.
myConstraintSolver	*ConstraintSolver*	Stores the instance of *ConstraintSolver* currently being used by the program.
GameOver	String	Stores whether the game is in progress, won or lost.
FirstMove	Boolean	Stores whether the first move has been made yet.
BoardStates	Char(,)	Stores the contents of each cell, each location in the array corresponds to the contents of the cell at that index.
CurrentStrategy	String	Stores the strategy used for the next move (Safe, Guess, Corner or Random), if the game is lost then this is passed to the live stats tracker to update the counter.
Guess	*BestGuessCell*	Stores the current *BestGuessCell*, to be used if no safe cells are found.
HardestAlgorithm	String	Stores the "hardest" algorithm needed to beat the current board so far, this is passed to the stats tracker at the end and used for training.

Methods

Name	Parameters	Purpose
PlayGame	N/A	The main routine for the solver, initiates the live stats, then begins the game by populating **Cells** and running **EvaluateBoard**. Once the board has been evaluated it is responsible for iterating through and opening all cells in **SafeCells** or calling **MakeInformedMove**.
OpenStatsForm	N/A	Responsible for deciding the size and location of the stats form on the screen, then displaying it.
EvaluateBoard	N/A	Routine responsible for managing the running of solving algorithms. It populates the attributes of each *Cell* in **Cells** then works through each algorithm in order, stopping if one of them populates

		SafeCells.
MakeInformedMove	N/A	Responsible for deciding the next move if **SafeCells** is empty. If **Guess** is available, it is used, and if not then an unopened corner is found using **FindCorner** and opened. If no corner is available a random cell is opened.
FindCorner	N/A	Checks each corner to see if it has been opened, if not then it is returned to **MakeInformedMove**.
SafeToClick	Cell (*Cell*)	Checks the that the cell passed to it is adjacent only to other cells with no information known about them, used in **ClickRandom** to ensure the cell is random.
EndGame	N/A	Runs when the game is finished, sends the data from that game to stats, then calls **ResetGame**.
GetCells	N/A	Populates **Cells** and returns it.
ClickRandom	Skip (*Boolean*)	Finds a random, non-mine cells in **Cells**, which no information is known about to click if no corners are available. If no cells pass **SafeToClick**, then **Cells** is iterated through again, but **SafeToClick** is skipped.

ConstraintSolver

Attributes

Name	Type	Purpose
Guess	*BestGuessCell*	Stores the current best guess, used only if **SafeCells** is empty.
CustomBoard	Boolean	Stores whether the board being solved was generated using the custom board function or not.
SafeCells	List(Of *Cell*)	Stores a dynamic list which is updated whenever a new safe cell is found.
OldCells	*Cell*(,)	Stores an unaltered version of the current board. This array is only updated if mines or safe cells are found.
Cells	*Cell*(,)	Starts of as an exact copy of

		OldCells. This array is allowed to be modified via reduction and when solving constraints.
EquationList	List(Of *CoupledEquation*)	Stores a list containing all the equations for the current board.
VariableList	List(Of *Variable*)	Stores a list containing all the variables for the current board.
AllSolutions	List(Of List(Of *Solution*))	Stores, for each item in **EquationList**, a list of the solutions to that equation.
SolutionCombos	List(Of *SolutionCombo*)	Stores a list containing every possible combination of solutions in **AllSolutions**.
TotalPermutations	Double	Stores the total number of possible permutations of the current board.
SolutionDict	Dictionary(Of Integer, List(Of *SolutionCombo*))	Stores a dictionary which contains every possible combination of solutions, organised according to the number of mines in each.
MineCells	Integer	Stores the total number of mines remaining on the board.
UnknownCells	Integer	Stores the total number of unknown cells remaining on the board.
ShortEval	Boolean	Stores whether the solver should use a quicker algorithm to evaluate the board and avoid a large amount of computation.

Methods

Name	Parameters	Purpose
New	CellArray (*Cell(,)*), SafeCellList (*List(Of Cell)*), Custom (*Boolean*)	Populates **OldCells** with *CellArray* and **Cells** with a clone of *CellArray* so changes can be made. Also populates **CustomBoard** and **SafeCells**.
FindNonTrivialSolutions	N/A	Reduces all cells in **Cells**, then compares each *Cell* with each other *Cell* to check if they share adjacents, and removes from their **AdjacentCells** accordingly. Determines if any safe cells or mines have now been revealed. If not then it forms equations and begins the brute force solving.
AddToEquation	Cell (*Cell*)	Searches through **EquationList**

		for a suitable *CoupledEquation* to add a constraint to, if one is not found or already has too many variables then a new one is made.
CombineEquations	N/A	Ran after all equations have been formed, compares each item in **EquationList** with every other one. If two *CoupledEquations* share common variables then they are combined (provided it doesn't exceed the limit of 20).
EvaluateEquationsLong	N/A	uses the brute force solver to generate and analyse solutions. If no safe cells are found then **SolutionCombos** is populated with every possible combination, and the global guessing algorithm is run. Finally it generates a best guess and compares it to a random cell using **RandomCellChance** to determine the best next move.
RandomCellChance	N/A	Determines the chance that a random cell is safe on the current board, and updates **Guess** if it is.
GetSafeCells	N/A	Getter for **SafeCells.**
GetGuess	N/A	Getter for **Guess.**

(NOTE THAT SOME METHODS HAVE NOT BEEN INCLUDED AS THEY ARE COPIED FROM THE INTERNET)

BruteForceSolver

Attributes

Name	Type	Purpose
CustomBoard	Boolean	Stores whether the board being solved was generated using the custom board function or not.
Cells	*Cell*(,)	Stores the current board, note that this is the modified version generated in the constraint solver.
SafeCells	List(Of *Cell*)	Stores a dynamic list which is updated whenever a new safe cell is found.

Equation	*CoupledEquation*	Stores the equation which is being solved.
VariableStack	Stack(Of *Variable*)	Stores the variables which are part of the *Solution* currently being generated.
ReturnToStack	Stack(Of *Variable*)	Stores the variables which need to be returned to and have their **Status** changed to generate the next *Solution*.
VariablesToCheck	List(Of *Variable*)	Stores a list of the variables in the current equations.
Solutions	List(Of *Solution*)	Stores a list of the solutions which have been generated to solve **Equation**.
SolutionFound	Boolean	Stores whether the current contents of **VariableStack** are a solution to **Equation**.

Attributes

Name	Parameters	Purpose
New	EquationToSolve (*CoupledEquation*), CellArray (*Cell(,)*), SafeCellList (*List(Of Cell)*), Custom (*Boolean*)	Populates **CustomBoard**, **Cells**, **SafeCells**, and **Equation**. Then orders the variables in **Equation** by their **Tally** and populates **VariablesToCheck** with them.
FindSolutions	N/A	Repeatedly calls **CreateStack**, then checks if a solution has been generated and adds to **Solutions** if so. Then pops **VariableStack** until then next *Variable* in **ReturnToStack** is found.
CreateStack	N/A	Pushes *Variables* to **VariableStack,** checking that their **Status** doesn't cause a contradiction in **Equation**.
GetMineCount	Variables (*List(Of Variable)*)	Returns the number of *Variables* which have a **Status** of 1.
DeduceSafeCells	N/A	Iterates through **Solutions** and checks for any consistencies which indicate a safe cell or a mine.
GetSolutions	N/A	Getter for **Solutions.**

Cell

Attributes

Names	Types	Purpose
Colour	String	Stores the hex code for the colour of the cell, this informs the number contained within it.
Contents	Char	Stores the Number contained within the cell, set to "u" if the cell is unknown.
Location	Point	Stores the coordinates relative to the grid of the upper left had pixel of the cell, used when clicking the cell.
Index	Point	Stores the index of the cell relative to the top left of the grid.
AdjacentCells	List(Of *Cell*)	Stores each *Cell* which is directly adjacent to the cell.
Useless	Boolean	Stores whether this cell should be considered when running algorithms.
Mine	Boolean	Stores whether this cell is a mine or not.
Occurrences	Double	Stores the number of permutations in which this cell is a mine.
ChanceSafe	Decimal	Stores the chance of this cell being safe, for use in guessing.

Methods

Name	Parameters	Purpose
New	Board (*Bitmap*), CellLocation (*Point*), CellIndex (*Point*),	Initiates the instance, sets both **Mine** and **Useless** to False and populates **Location, Index, Colour** (by calling **GetCellColour**) and **Contents** (by calling **GetCellState**). This constructor is only called from custom board functions.
New	State (*Char*), ArrIndex (*Point*)	Initiates the instance, sets both **Mine** and **Useless** to False and populates **Index, Location** (16x **Index**) and **Contents**
GetCellColour	Board (*Bitmap*)	Populates **Colour** with the HTML code for the colour of a pixel in the cell, used to identify its contents.
GetCellState	N/A	Uses **Colour** to populate **Contents** with the contents of the cell.
CheckIfUseless	N/A	Determine whether this *Cell* is

		Useless by checking if it is completely surrounded by mines.
GenerateAdjacents	Cells (*Cells(,)*), BoardWidth (*Integer*), BoardHeight (*Integer*)	Populates **AdjacentCells** with each *Cell* that is directly adjacent on the board.
CheckIfMine	N/A	Compares the number of items in **AdjacentCells** with **Contents**, and sets **Mine** to true in all **AdjacentCells** if they are equal.
SimpleFinder	N/A	Runs the simple finder algorithm detailed above, returns any safe cells it finds for opening.
ReduceCell	N/A	Removes all mines from **AdjacentCells** and decrements **Contents** by 1 for each.
CountSharedAdjacents	OtherCell (*Cell*)	Checks through the **AdjacentCells** of *OtherCell*, and returns how many are shared between the two cells for .
RemoveSharedAdjacents	OtherCell (*Cell*)	Removes any adjacents that are shared between the two cells from **AdjacentCells** of *OtherCell*. Decrements **Contents** accordingly.
UpdateMines	SafeCells (*List(Of Cell)*)	Checks **AdjacentCells** against *SafeCells*. If there are enough safe cells in **AdjacentCells** then the remaining will have **Mine** set to true.
GetOccurences	N/A	Getter/Setter for **Occurrences.**
GetChanceSafe	N/A	Getter/Setter for **ChanceSafe..**
CellContents	N/A	Getter/Setter for **Contents**
Adjacents	N/A	Getter for **Adjacents.**
IsUseless	N/A	Getter/Setter for **Useless.**
IsMine	N/A	Getter/Setter for **Mine.**
GetIndex	N/A	Getter for **Index.**
GetLocation	N/A	Getter for **Location.**

Variable

Attributes

Name	Type	Purpose
Cell	*Cell*	Stores the *Cell* associated with this *Variable*.
Tally	Integer	Stores the number of times this variable appears in a *Constraint*.

Status	Integer	Stores either 0 or 1 depending on whether this *Variable* is a mine or not.

Methods

Name	Parameters	Purpose
New	NewCell	Initiates instance, populates **Cell** and sets both **Tally** and **Status** to 0.
GetCell	N/A	Getter for **Cell.**
GetTally	N/A	Getter for **Tally.**
GetStatus	N/A	Getter for **Status.**

Constraint

Attributes

Name	Type	Purpose
Variables	List(Of *Variable*)	Stores a list containing the variables that this constraint acts on.
Mines	Integer	Stores the number of mines contained within the constraint.

Methods

Name	Parameters	Purpose
New	Cell (*Cell*), NewVariables (*List (Of Variable)*), EquationList(*List (Of CoupledEquation)*)	Populates **Variables** and **Mines** using the **Contents** of *Cell* and *NewVariables*.
IsSatisfied	N/A	Returns True if the number of items in **Variables** with a **Status** of 1 is equal to **Mines**. Returns false otherwise.
GetVariables	N/A	Getter for **Variables**.
GetTotalMines	N/A	Getter for **Mines.**

CoupledEquation

Attributes

Name	Type	Purpose
Solutions	List(Of *Solution*)	Stores a list of the solutions to

		the constraints.
Variables	List(Of *Variable*)	Stores a list of the variables included in the constraints.
Constraints	List(Of *Constraint*)	Stores a list of the constraints that **Variables** are subject to.

Methods

Name	Parameters	Purpose
New	Cell (*Cell*), EquationList (*List(Of CoupledEquation)*), VariableList (*List(Of Variable)*)	Populates **Variables** with a set of variables corresponding to *Cell*'s **AdjacentCells**. Then populates **Constraints** with a new *Constraint* for **Variables**
AddVariable	Cell (*Cell*), EquationList (*List(Of CoupledEquation)*), VariableList (*List(Of Variable)*)	Searches through the **AdjacentCells** of *Cell*, appending to **Variables** for any cells whose *Variable* hasn't already been added. Then appends to **Constraints** with a new constraint on **Variables**.
GetVariables	N/A	Getter/Setter for **Variables**.
GetConstraints	N/A	Getter for **Constraints.**

Solution

Attributes

Name	Type	Purpose
Variables	List(Of *Variable*)	Stores each *Variable* in the solution.
Mines	Integer	Stores the number of mines in the solution.

Methods

Name	Parameters	Purpose
New	VariablesInSolutions	Initiates the instance and populates **Variables**.
AddOccurences	NumOfOccurences (*Double*), OldCells (*Cell(,)*)	Iterates though each *Variable* in **Variables**. If it is a mine, then that variables respective *Cell* has **NumOfOccurences** added to its **Occurrences**.
GetVariables	N/A	Getter for **Variables**.
GetMines	N/A	Getter for **Mines.**

SolutionCombo

Attributes

Name	Type	Purpose
Solutions	List(Of *Solution*)	Stores each *Solution* in the combo
NumberOfOccurences	Double	Stores the number of permutations which use this combination.
NumberOfMines	Integer	Stores the number of mines in this combination.
MinesLeftOnBoard	Integer	Stores the number of mines still left on the board whose locations are unknown, not including the mines accounted for by **NumberOfMines**.
CellsLeftOnBoard	Integer	Stores the number of cells left on the board whose contents are unknown, not including the ones which are accounted for in **Solutions**.

Methods

Name	Parameters	Purpose
New	SolutionList	Initiates the instance and populates **Solutions**.
CountMines	N/A	Iterates through **Solutions**, to total up the number of mines in each, then populates **NumberOfMines** with this value.
FindPermutations	OldCells (*Cell(,)*), MineCells (*Integer*), UnknownCells (*Integer*), CustomBoard (*Boolean*)	Calculates and populates **MinesLeftOnBoard** and **CellsLeftOnBoard**. Then uses the choose function (nCr) to populate **NumberOfOccurences**.
Pascal	n (*Integer*), r (*Integer*)	Performs the choose function (or finds the correct value in Pascal's Triangle) for the values of **n** and **r** and returns it.
GetPermutations	OldCells (*Cell(,)*)	Getter/Setter for **NumberOfOccurences.** If used as a setter, then calls **AddOccurences** for each item in **Solutions** as well.
CellsLeft	N/A	Getter/Setter for **CellsLeftOnBoard.**

MinesLeft	N/A	Getter/Setter for **MinesLeftOnBoard.**
GetMines	N/A	Iterates through **Solutions** and tallies the number of mines in each, then returns this value.

BestGuessCell

Attributes

Name	Type	Purpose
Cell	*Cell*	Stores the cell associated with the current best guess.
ChanceSafe	Decimal	Stores the probability that the best guess is safe.
Used	Boolean	Stores whether the guess was used and opened or not.

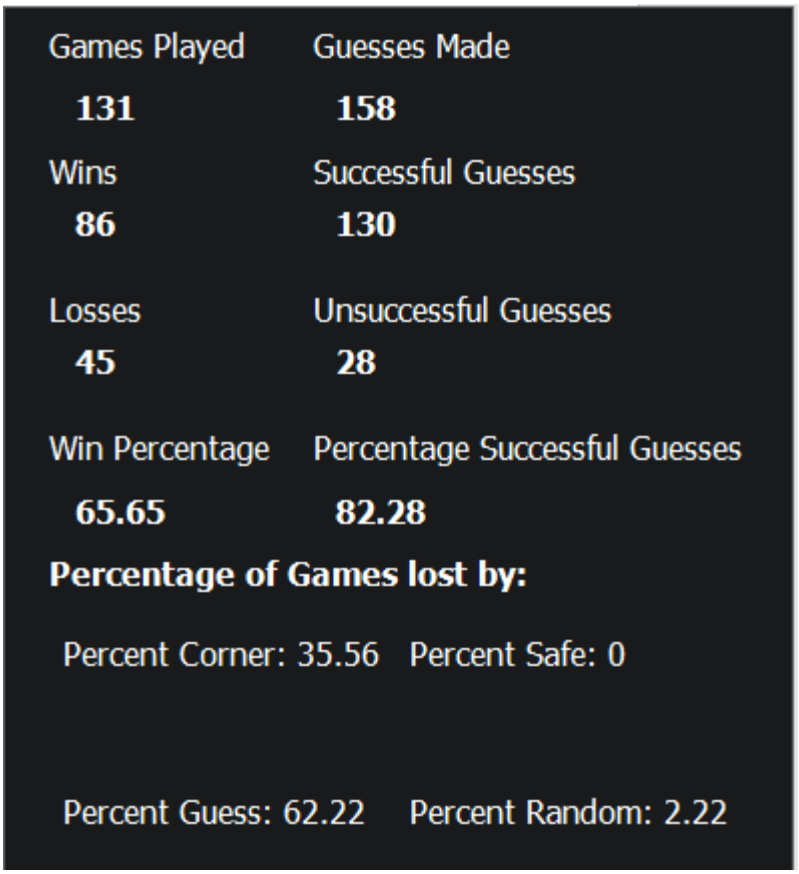
Methods

Name	Parameters	Purpose
BestGuess	NewGuess (*Cell*), NewChance (*Decimal*)	Compares the *NewChance* with **ChanceSafe** to determine if *NewCell* is safer than **Cell**. If so then they are updated.
RandomCell	Chance (*Decimal*)	Called if a random cell has been determined to be the best guess. Empties **Cell** and updates **ChanceSafe**.
ShortEval	Cells (*Cell(,)*)	Iterates through *Cells* and finds the safest cell, and then populates **Cell** and **ChanceSafe** with its information.
GetCell	N/A	Getter for **Cell.**
GetChanceSafe	N/A	Getter for **ChanceSafe.**
GetUsed	N/A	Getter/Setter for **Used.**

Stats

As either the human or computer plays, several pieces of information are collected and stored, or displayed to the user. This is done in two separate places, Live Stats and Overall Stats.

Live Stats



The Live Stats section of the program was a feature requested by my end user, this window is displayed whenever the computer is playing. After every game, a call is made to update the window with the data from that game. Here is a video of this in action[3].

Throughout the game, calls are made to this window whenever a guess is made. This will increment the **Guesses Made** and then also the **Successful Guesses** or **Unsuccessful Guesses** depending on the outcome of that guess.

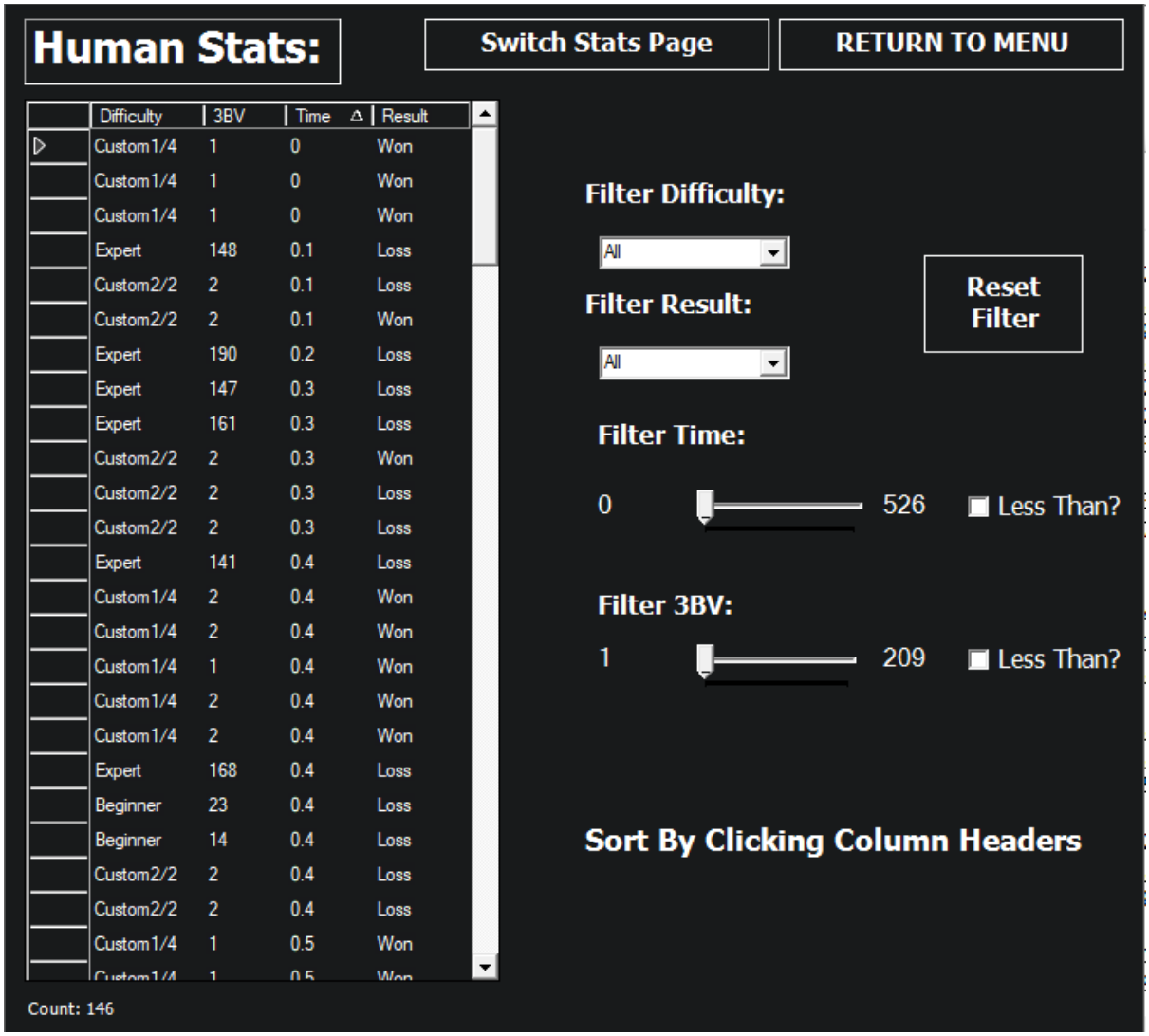
After the game, a call is made with the result of the game. When this is received, the **Games Played** counter increments, and then either **Wins** or **Losses** increments depending on the result.

If the game was lost, then the Live Stats will also be told what kind of cell was clicked to lose it. This data is stored in a table, which records how many times each of the 4 strategies have been the cause of a loss, and percentages are generated by dividing the **Count** for one by the sum of the **Count** for all 4. For example, in the picture above the table would look like this:

Strategy	**Count**
Safe A cell which has been deduced as being 100% safe (should always be 0)	**0**
Guess A cell which has been decided as the best guess	**28**
Corner A cell in the corner, which has been decided to be a better alternative to a guess	**16**

Random A cell which has been decided to be a better alternative to guess, when no corners are available	**1**

Overall Stats



The Overall Stats are recorded for both human players and the computer. After each game, the following data is recorded and stored:

- **Difficulty** – The difficulty of the game (Expert, Intermediate, Beginner or Custom).
- **3BV** – the 3BV of the board, which is calculated as the board is being generated.
- **Time** – the time taken to finish the board (whether it was won or lost).
- **Result** – Whether the game was won or lost.
- **Algorithm** – The algorithm furthest down the hierarchy which was used to beat the board (Only used for computer games)

- **Board –** a tailor made format for storing a Minesweeper board so it can be recreated (more detail later)

After each game, a string in a csv format is created and stored in a text file (**HumanStats** or **ComputerStats**). When the stats window is opened, the contents of this file are read into two separate tables stored in memory. The relevant information from the tables for each game are then displayed on the screen to the user.

As two tables are used, for Human and Computer respectively, their record of games can be viewed separately and switched between using the **Switch Stats Page** button.

Filters

For a better user experience, filters are available to update the table and have it show only games that comply to the filters.

To allow for filtering, a **Filter** parent class is used, along with two child classes, ***StringFilter*** and ***NumericFilter***.

- **Difficulty** – String Filter
- **Result** – StringFilter
- **Time** – NumericFilter
- **3BV** - NumericFilter

Each filter has its own **FilterExpression** which can be applied to the table so it complies with that filter, this expression uses syntax which is built in to the .NET framework specifically for filtering tables[13]. Each ***NumericFilter*** also has a **FilterBar** and **LessThanCheckbox** whilst each ***StringFilter*** has an **OptionsBox**.

When a filter is chosen by the user, the **UpdateFilter** routine is called. For a ***NumericFilter***, this can change the **FilterExpression** to one of two things using this pseudocode:

```
If LessThanChecked Then
      FilterExpression = "[" & FilterName & "] <= " & FilterBarValue
Else
      FilterExpression = "[" & FilterName & "] >= " & FilterBarValue
End If
```

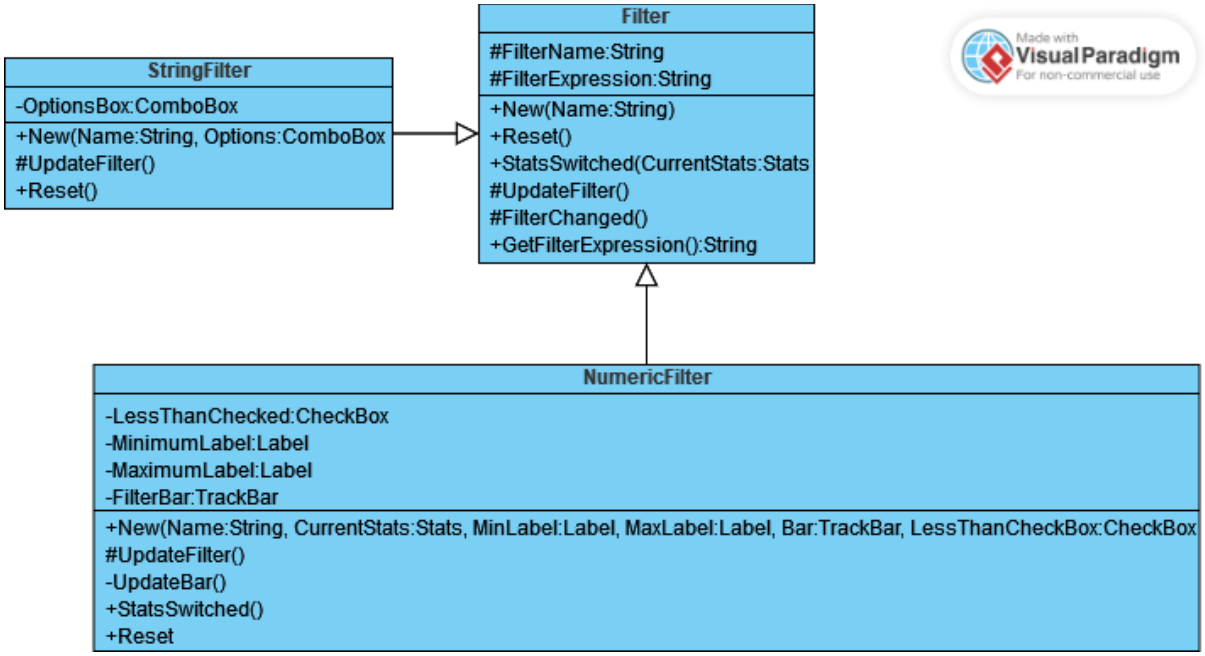
For a ***StringFilter***, it is changed using this pseudocode:

```
If OptionsBoxItem <> "All" Then
      FilterExpression = FilterName & "LIKE '*" & OptionsBoxItem & "*'"
Else
      FilterExpression = ""
End If
```

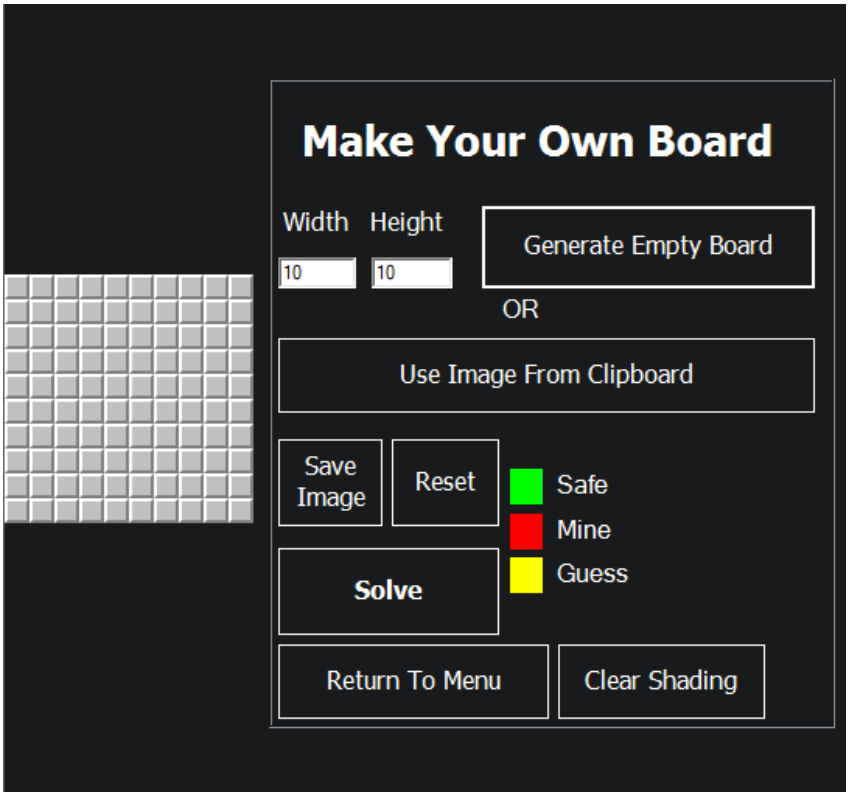
LIKE is used to allow for all custom sized boards to be selected with one filter. As the board dimensions are displayed as part of the difficulty for a custom sized board, they are not all identical. The **LIKE** expression, combined with two wildcards around the name (*), allow for any boards containing the word “Custom” to be filtered as one.

Both versions of the routine then raise an event called **FilterChanged**, which is handled by the table. The table collects together the **FilterExpression** from each filter, and combines them using an “and” operator into one expression, which is then applied to the table and displayed to the user.

Here is a class diagram to show the relationship between the **Filter** class and its children:



Custom Boards

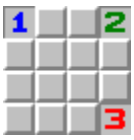


The Custom Board section of the program works very similarly to and utilises many of the same algorithms as the solver. The user is presented with a blank grid of cells, which they can edit the contents of manually by either clicking the cell and choosing the value, or hovering over it and pressing the corresponding key on the keyboard (A video of this can be found here[4]). Here are the keys for editing contents:

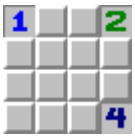
Key	Contents
1, 2, 3, 4, 5, 6, 7, 8	1, 2, 3, 4, 5, 6, 7, 8
0, Space	Blank
U	Unknown/Unopened Cell
F, M	Flag

The **Solve** button can then be used, and the computer will attempt to solve the board. It will only complete the solving algorithms if the board is valid. There are two tests which can be done to ensure a board is valid:

Test 1 - Is the number of adjacents for each cell less than the number in the cell?



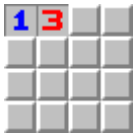
Valid



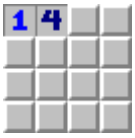
Invalid

In the example above, we can see that in the bottom right corner of the valid board is a three, which is adjacent to three cells, meaning it conforms to the statement. The invalid board, however, has a 4 bordering those same three cells, this 4 cannot be placed there and thus the board is invalid.

Test 2 - After solving, are the number of cells which could still be mines less than the number in the cell?



Valid



Invalid

In this example, both boards would pass **Test 1**, although after the computer begins to solve this board, it would find a problem with the invalid board. Due to the 1 in the top left corner, only three mines can possibly be in the cells adjacent to the 4, meaning the 4 cannot be placed there and thus the board is invalid.

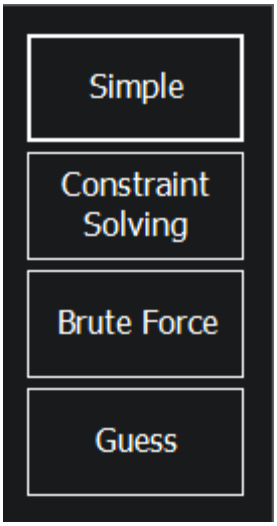
Both tests use the same line of code, although **Test 1** runs before any computation is carried out, and **Test 2** runs after each cell has been reduced when going through the constraint solver, as the numbers within each cell and their adjacents change during this process. Here is the pseudocode for the tests:

```
For Each Cell
      If NumberOfAdjacentCells < NumberInCell Then
            Board is Invalid
      End If
Next
```

If a flag is placed on the board, the solver will treat that cell as being a guaranteed mine and will take this into account when determining validity and when finding safe cells/guesses.

One thing to note about the custom boards is that the user does not specify how many mines are present in the board. For this reason, if a guess is required to solve the board, the computer will always use local guessing. This means that when comparing the solved board from this function to one in an actual game of Minesweeper, the best guess may vary. The decision not to include a mine count was made for ease of use to the user.

Trainer



The final function of my program is the trainer, another feature requested by the end user. The menu contains 4 buttons, each of which correspond to an algorithm used by my solver. When the user clicks one of these buttons, a random board will be given to the user. This board will be solvable without using any technique more complicated than the one chosen by the user, for example if the user clicked on Constraint Solving, then a board would be generated that the computer could solve using only its constraint solving or simple finder algorithms.

In order to generate a new board every time, of which the difficulty is known, the stats file is used. When a button is pressed, the computer will search randomly through the games in the ComputerStats file until it finds one whose "Algorithm" column matches the button pressed, and then uses the "Board" column to recreate that game.

In order to recreate a game, the "Board" column of a game in stats is formatted in a specific way:

9 9 10 0,2 0,3 2,1 2,2 3,2 3,3 4,2 4,7 6,5 7,0

This is the width of the board.

This is the height of the board.

This is the number of mines on the board.

This is the location of each mine on the board as a coordinate in the form x,y (0,0 is located in the top left of the board)

The computer will read this board into memory and recreate it, then display it to the user as a playable game. This board recreation can also be done on any game in stats by double clicking it in the table in the stats window.

Technical Solution

The Technical Solution is broken down into several Forms/Modules, each with its own functionality:

(Note that any highlighted code was taken from the Internet)

MenuForm

```
Class MenuForm


    Private Sub PlayButton_Click(sender As Object, e As EventArgs) Handles
PlayButton.Click
        Hide()
        myGameplayForm.Show()
    End Sub
    Private Sub StatsButton_Click(sender As Object, e As EventArgs) Handles
StatsButton.Click
        Hide()
        myStatsForm = New OverallStatsForm
        myStatsForm.Show()
    End Sub
    Private Sub CustomBoardButton_Click(sender As Object, e As EventArgs) Handles
CustomBoardButton.Click
        Hide()
        myCustomGameForm = New CustomBoardForm
        myCustomGameForm.Show()
    End Sub

    Private Sub ExitButton_Click(sender As Object, e As EventArgs) Handles
ExitButton.Click
        End
    End Sub


    Private Sub TrainerButton_Click(sender As Object, e As EventArgs) Handles
TrainerButton.Click
        Hide()
        myTrainerForm = New TrainerForm
        myStatsForm = New OverallStatsForm
        myTrainerForm.Show()
        myGameplayForm.Show()
    End Sub




    Public MoveForm As Boolean
    Public MoveForm_MousePosition As Point

    Public Sub MoveForm_MouseDown(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseDown ' Add more handles here (Example: PictureBox1.MouseDown)

        If e.Button = MouseButtons.Left Then
            MoveForm = True
            Me.Cursor = Cursors.NoMove2D
            MoveForm_MousePosition = e.Location
```

```
        End If

    End Sub

    Public Sub MoveForm_MouseMove(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseMove ' Add more handles here (Example: PictureBox1.MouseMove)

        If MoveForm Then
            Me.Location = Me.Location + (e.Location - MoveForm_MousePosition)
        End If

    End Sub

    Public Sub MoveForm_MouseUp(sender As Object, e As MouseEventArgs) Handles _
    MyBase.MouseUp ' Add more handles here (Example: PictureBox1.MouseUp)

        If e.Button = MouseButtons.Left Then
            MoveForm = False
            Me.Cursor = Cursors.Default
        End If

    End Sub
End Class
```

GameplayForm

```
Imports System.IO
Public Class GameplayForm

    Private Board As GameBoard
    Public Stopped As Boolean = False
    Private ComputerPlaying As Boolean
    Private Time As Decimal
    Private Random As Boolean = False
    Private TimerThread As Threading.Thread
    Private GameEnded As Boolean
    Private ChangeBoard As String
    Private Sub GameplayForm_Load(sender As Object, e As EventArgs) Handles
MyBase.Load
        Board = New GameBoard(30, 16, 99)
        BoardBox.Image = Board.GetBoard
        TimeBox.Text = 0
        ChangeBoard = "ExpertButton"
        GenerateNewSizeBoard()
    End Sub

    Sub GameFromStats(States As String)
        Show()
        Board = New GameBoard(States)
        BoardBox.Image = Board.GetBoard
        BoardBox.Size = BoardBox.Image.Size
        ResizeForm()
        TimeBox.Text = 0
    End Sub

    Public Sub Form_Unload() Handles Me.Closing
        myMenuForm.Show()
    End Sub

    Sub ResizeForm()
```

```
        'makes sure the window size is proportional to the board size
        UserOptionsBox.Location = New Point(BoardBox.Location.X + BoardBox.Width
+ 10, UserOptionsBox.Location.Y)
        Size = New Size(UserOptionsBox.Location.X + UserOptionsBox.Width + 10,
Math.Max(UserOptionsBox.Height + 43, BoardBox.Height + 43) + 43)
        BoardBox.Location = New Point(0, (Size.Height - BoardBox.Height) / 2)
        UserOptionsBox.Location = New Point(UserOptionsBox.Location.X,
(Size.Height - UserOptionsBox.Height) / 2)
    End Sub

    Private Sub NewBoard(sender As Button, e As EventArgs) Handles
GenerateBoardButton.Click, BeginnerButton.Click, IntermediateButton.Click,
ExpertButton.Click
        If sender.Name = "GenerateBoardButton" Then ChangeBoard = "Custom" Else
ChangeBoard = sender.Name
        Select Case ChangeBoard
            Case "BeginnerButton"
                Board = New GameBoard(9, 9, 10)
            Case "IntermediateButton"
                Board = New GameBoard(16, 16, 40)
            Case "ExpertButton"
                Board = New GameBoard(30, 16, 99)
            Case "Custom"
                If Not (XSizeBox.Value = 0 Or YSizeBox.Value = 0 Or
XSizeBox.Value > 50 Or YSizeBox.Value > 50 Or MinesBox.Value = 0 Or
MinesBox.Value >= XSizeBox.Value * YSizeBox.Value) Then
                    Board = New GameBoard(XSizeBox.Value, YSizeBox.Value,
MinesBox.Value)
                End If
        End Select
        BoardBox.Image = Board.GetBoard
        BoardBox.Size = BoardBox.Image.Size
        ResizeForm()
    End Sub

    Private Sub GenerateNewSizeBoard()
        Select Case ChangeBoard
            Case "BeginnerButton"
                Board = New GameBoard(9, 9, 10)
            Case "IntermediateButton"
                Board = New GameBoard(16, 16, 40)
            Case "ExpertButton"
                Board = New GameBoard(30, 16, 99)
            Case "Custom"
                If Not (XSizeBox.Value = 0 Or YSizeBox.Value = 0 Or
XSizeBox.Value > 50 Or YSizeBox.Value > 50 Or MinesBox.Value = 0 Or
MinesBox.Value >= XSizeBox.Value * YSizeBox.Value) Then
                    Board = New GameBoard(XSizeBox.Value, YSizeBox.Value,
MinesBox.Value)
                End If
        End Select
        BoardBox.Image = Board.GetBoard
        BoardBox.Size = BoardBox.Image.Size
        ResizeForm()
    End Sub

    Private Sub OpenCellClick(sender As Object, e As MouseEventArgs) Handles
BoardBox.Click
        If Board.GetEndResult = "" Then
            If e.Button = MouseButtons.Left Then Board.UpdateBoard(e.Location,
True) Else Board.UpdateBoard(e.Location, False)
            If Board.GetEndResult = "Won" Then ResetButton.Image =
My.Resources.ResetButtonWon : AbortTimer(False)
```

```
            If Board.GetEndResult = "Loss" Then ResetButton.Image =
My.Resources.ResetButtonLost : AbortTimer(False)
            BoardBox.Image = Board.GetBoard
            ComputerPlaying = False
            If Board.GetEndResult <> "" Then WriteStatsToFile()
        End If
    End Sub

    Public Sub OpenCellProgram(Location As Point)
        If Board.GetEndResult = "" Then
            Board.UpdateBoard(Location, True)
            If Board.GetEndResult = "Won" Then ResetButton.Image =
My.Resources.ResetButtonWon : AbortTimer(False)
            If Board.GetEndResult = "Loss" Then ResetButton.Image =
My.Resources.ResetButtonLost : AbortTimer(False)
            BoardBox.Image = Board.GetBoard
            ComputerPlaying = True
            If Board.GetEndResult <> "" Then WriteStatsToFile()
            Application.DoEvents()
        End If

    End Sub

    Public Sub ResetGame() Handles ResetButton.Click
        GenerateNewSizeBoard()
        If Not Random Then
            Board = New GameBoard(Board.GetBoard.Width / 16,
Board.GetBoard.Height / 16, Board.GetMines)
        Else
            Dim r As New Random
            Dim Width As Integer = r.Next(9, 31)
            Dim Height As Integer = r.Next(9, 31)
            Dim Mines As Integer = r.Next(10 / 81 * Width * Height, 99 / 480 *
Width * Height + 1) ' makes sure the board is somewhere between the difficulty of
a beginner and expert board
            Board = New GameBoard(Width, Height, Mines)
        End If
        BoardBox.Image = Board.GetBoard
        BoardBox.Size = BoardBox.Image.Size
        ResizeForm()
        Board.IsFirstMove = True
        ResetButton.Image = My.Resources.ResetButton
        AbortTimer(True)
    End Sub

    Private Sub ReturnToMenuButton_Click(sender As Object, e As EventArgs)
Handles ReturnToMenuButton.Click
        ResetGame()
        Stopped = True
        Hide()
        If myLiveStatsForm IsNot Nothing Then myLiveStatsForm.Hide()
        If myTrainerForm IsNot Nothing Then myTrainerForm.Hide()
        myMenuForm.Show()
    End Sub

    Private Sub WriteStatsToFile()
        Dim Writer As StreamWriter
        If ComputerPlaying Then
            Writer = New StreamWriter(Directory.GetCurrentDirectory +
"\ComputerStats.txt", True)
            Writer.WriteLine(Board.GetDifficulty & "," & Board.Get3BV & "," &
TimeBox.Text & "," & Board.GetEndResult & "," & myGame.HardestAlgorithm & "," &
Board.GetStates)
```

```
        Else
            Writer = New StreamWriter(Directory.GetCurrentDirectory +
"\HumanStats.txt", True)
            Writer.WriteLine(Board.GetDifficulty & "," & Board.Get3BV & "," &
TimeBox.Text & "," & Board.GetEndResult & "," & "" & "," & Board.GetStates)
        End If
        Writer.Close()
    End Sub

    Public Function GetBoardStates() As Char(,)
        If Board.GetEndResult <> "" Then myGame.GameOver = Board.GetEndResult :
Return Nothing
        Dim States(Board.GetCells.GetUpperBound(0),
Board.GetCells.GetUpperBound(1)) As Char
        For x = 0 To Board.GetCells.GetUpperBound(0)
            For y = 0 To Board.GetCells.GetUpperBound(1)
                If Board.GetCells(x, y).IsOpen Then
                    If Board.GetCells(x, y).GetState = "m" Then Return Nothing
                    States(x, y) = Board.GetCells(x, y).GetState
                    myGame.FirstMove = False
                Else
                    States(x, y) = "u"
                End If
            Next
        Next
        Return States
    End Function

    Private Sub PlayButton_Click(sender As Object, e As EventArgs) Handles
PlayButton.Click
        If ComputerPlaying Then Stopped = False : myGame.PlayGame() : Exit Sub
        Stopped = False
        myGame.PlayGame()
    End Sub

    Private Sub ReplayButton_Click(sender As Object, e As EventArgs) Handles
ReplayButton.Click
        Board.Reset()
        AbortTimer(True)
        TimeBox.Text = 0
        BoardBox.Image = Board.GetBoard
    End Sub

    Private Sub RandomPlayButton_Click(sender As Object, e As EventArgs) Handles
RandomPlayButton.Click
        If Random = False Then
            Random = True
            RandomPlayButton.BackColor = ColorTranslator.FromHtml("#4AF26B")
        Else
            Random = False
            RandomPlayButton.BackColor = ColorTranslator.FromHtml("#F24A4A")
        End If
    End Sub

    Private Sub StopButton_Click(sender As Object, e As EventArgs) Handles
StopButton.Click
        Stopped = True
    End Sub

    Sub AbortTimer(Clear As Boolean)
        If TimerThread IsNot Nothing Then TimerThread.Abort()
        If Clear Then TimeBox.Text = 0
    End Sub
```

```
    Sub NewTimer()
        If TimerThread IsNot Nothing Then TimerThread.Abort()
        TimerThread = New Threading.Thread(AddressOf TimerHandler)
        TimerThread.Start()
    End Sub
    Sub TimerHandler()
        Dim sleep As New Threading.ManualResetEvent(False)
        TimeBox.Invoke(Sub() TimeBox.Text = 0)
        Do
            sleep.WaitOne(100)
            TimeBox.Invoke(Sub() TimeBox.Text += 0.1)
        Loop
    End Sub


    ReadOnly Property GetMines As Integer
        Get
            Return Board.GetMines
        End Get
    End Property
```

```
    Public MoveForm As Boolean
    Public MoveForm_MousePosition As Point

    Public Sub MoveForm_MouseDown(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseDown ' Add more handles here (Example: PictureBox1.MouseDown)

        If e.Button = MouseButtons.Left Then
            MoveForm = True
            Me.Cursor = Cursors.NoMove2D
            MoveForm_MousePosition = e.Location
        End If

    End Sub

    Public Sub MoveForm_MouseMove(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseMove ' Add more handles here (Example: PictureBox1.MouseMove)

        If MoveForm Then
            Me.Location = Me.Location + (e.Location - MoveForm_MousePosition)
        End If

    End Sub

    Public Sub MoveForm_MouseUp(sender As Object, e As MouseEventArgs) Handles _
    MyBase.MouseUp ' Add more handles here (Example: PictureBox1.MouseUp)

        If e.Button = MouseButtons.Left Then
            MoveForm = False
            Me.Cursor = Cursors.Default
        End If

    End Sub
End Class
```

```
<Serializable> Public Class GameBoard

    Private Board As Bitmap
    Private CellRects As Rectangle(,)
    Private GameCells As GameCell(,)
    Private MineCount As Integer
    Private OpenedCount As Integer
    Private Ended As String
    Private FirstMove As Boolean
    Private _3BV As Integer
    Private Difficulty As String
    Sub New(Width As Integer, Height As Integer, Mines As Integer)
        MineCount = Mines
        ResetGame(Width, Height)
    End Sub

    Sub ResetGame(Width As Integer, Height As Integer)
        NewGame(Width, Height)
        GenerateMines()
        For Each Cell In GameCells
            If Cell.GetState <> "m" Then Cell.GenerateAdjacentCells(GameCells,
New Point(GameCells.GetUpperBound(0), GameCells.GetUpperBound(1)))
        Next
        Generate3BV()
    End Sub

    Sub New(States As String)
        ' generating a board from stats using the mine locations
        Dim Width As Integer
        Dim Height As Integer
        Dim Index As Integer
        Dim x As Integer
        Dim y As Integer
        Do
            Width = Width & States(Index)
            Index += 1
        Loop Until States(Index) = " "
        Do
            Height = Height & States(Index)
            Index += 1
        Loop Until States(Index) = " "
        Index += 1
        Do
            MineCount = MineCount & States(Index)
            Index += 1
        Loop Until States(Index) = " "
        Index += 1
        ReDim GameCells(Width - 1, Height - 1)
        NewGame(Width, Height)
        Do
            Do
                x = x & States(Index)
                Index += 1
            Loop Until States(Index) = ","
            Index += 1
            Do
                y = y & States(Index)
                Index += 1
            Loop Until States(Index) = " "
            Index += 1
            GameCells(x, y).GetState = "m"
            x = Nothing
```

```
            y = Nothing
        Loop Until Index = States.Length

        For Each Cell In GameCells
            If Cell.GetState <> "m" Then Cell.GenerateAdjacentCells(GameCells,
New Point(GameCells.GetUpperBound(0), GameCells.GetUpperBound(1)))
        Next

        Generate3BV()
    End Sub

    Sub NewGame(Width As Integer, Height As Integer)

        ReDim GameCells(Width - 1, Height - 1)
        ReDim CellRects(Width - 1, Height - 1)

        If Width = 30 AndAlso Height = 16 AndAlso MineCount = 99 Then
            Difficulty = "Expert"
        ElseIf Width = 16 AndAlso Height = 16 AndAlso MineCount = 40 Then
            Difficulty = "Intermediate"
        ElseIf Width = 9 AndAlso Height = 9 AndAlso MineCount = 10 Then
            Difficulty = "Beginner"
        Else
            Difficulty = "Custom" & Width & "/" & Height
        End If

        Ended = ""
        FirstMove = True
        OpenedCount = 0

        Board = New Bitmap(Width * 16, Height * 16)
        Using g = Graphics.FromImage(Board)
            For x = 0 To CellRects.GetUpperBound(0)
                For y = 0 To CellRects.GetUpperBound(1)
                    CellRects(x, y) = New Rectangle(x * 16, y * 16, 15, 15)
                    g.DrawImage(My.Resources.UnknownCell, x * 16, y * 16)
                Next
            Next
        End Using

        For y = 0 To GameCells.GetUpperBound(1)
            For x = 0 To GameCells.GetUpperBound(0)
                GameCells(x, y) = New GameCell(New Point(x, y))
            Next
        Next

    End Sub


    Function GetStates() As String
        Dim States As String = GameCells.GetUpperBound(0) + 1 & " " &
GameCells.GetUpperBound(1) + 1 & " " & MineCount & " "
        For Each Cell In GameCells
            If Cell.GetState = "m" Then States += Cell.GetIndex.X & "," &
Cell.GetIndex.Y & " "
        Next
        Return States
    End Function

    Sub Reset()
        Using g = Graphics.FromImage(Board)
            For x = 0 To CellRects.GetUpperBound(0)
                For y = 0 To CellRects.GetUpperBound(1)
```

```
                    CellRects(x, y) = New Rectangle(x * 16, y * 16, 15, 15)
                    g.DrawImage(My.Resources.UnknownCell, x * 16, y * 16)
                Next
            Next
        End Using
        For Each Cell In GameCells
            Cell.IsOpen = False
            Cell.IsFlagged = False
        Next
        Ended = ""
        FirstMove = True
        OpenedCount = 0
    End Sub

    Private Sub Generate3BV()
        _3BV = 0
        For Each Cell In GameCells
            If Not Cell.IsMarked Then
                If Cell.GetState = "0" Then
                    _3BV += 1
                    Cell.IsMarked = True
                    Mark0s(Cell)
                End If
            End If
        Next
        For Each Cell In GameCells
            If Not Cell.IsMarked AndAlso Not Cell.GetState = "m" Then
Cell.IsMarked = True : _3BV += 1
        Next
    End Sub

    Sub Mark0s(Cell As GameCell)
        For Each Adjacent In Cell.GetAdjacents
            If Not Adjacent.IsMarked Then
                Adjacent.IsMarked = True
                If Adjacent.GetState = "0" Then Mark0s(Adjacent)
            End If
        Next
    End Sub

    Sub UpdateBoard(Location As Point, LeftClick As Boolean)
        ' opens the cell which has been clicked
        Using g = Graphics.FromImage(Board)
            For Each Rect In CellRects
                If Rect.Contains(Location) Then
                    If LeftClick Then
                        If Not GameCells(Rect.X / 16, Rect.Y / 16).IsOpen And Not
GameCells(Rect.X / 16, Rect.Y / 16).IsFlagged Then
                            Threading.Thread.Sleep(myGameplayForm.DelayBox.Value)
                            OpenCell(Rect.Location)
                        Else
                            If GameCells(Rect.X / 16, Rect.Y / 16).GetState <>
"0" Then
                                If GameCells(Rect.X / 16, Rect.Y /
16).GetAdjacents.FindAll(Function(x) x.IsFlagged).Count = Val(GameCells(Rect.X /
16, Rect.Y / 16).GetState) Then
                                    For Each Adjacent In GameCells(Rect.X / 16,
Rect.Y / 16).GetAdjacents
                                        If Not Adjacent.IsFlagged And Not
Adjacent.IsOpen Then OpenCell(Adjacent.GetLocation)
                                    Next
                                End If
                            End If
```

```
                        End If
                    Else
                        If Not GameCells(Rect.X / 16, Rect.Y / 16).IsOpen Then
                            If Not GameCells(Rect.X / 16, Rect.Y / 16).IsFlagged
Then
                                g.DrawImage(My.Resources.FlagCell, Rect.Location)
                                GameCells(Rect.X / 16, Rect.Y / 16).IsFlagged =
True
                            Else
                                g.DrawImage(My.Resources.UnknownCell,
Rect.Location)
                                GameCells(Rect.X / 16, Rect.Y / 16).IsFlagged =
False
                            End If
                        End If
                    End If
                End If
            Next
        End Using
    End Sub

    Sub OpenCell(Location As Point)
        Using g = Graphics.FromImage(Board)
            If FirstMove Then myGameplayForm.NewTimer()
            If Ended = "" Then
                GameCells(Location.X / 16, Location.Y / 16).IsOpen = True
                '     OpenedCount += 1
                Select Case GameCells(Location.X / 16, Location.Y / 16).GetState
                    Case "0"
                        g.DrawImage(My.Resources.BlankCell, Location)
                        '   If Not GameCells(Location.X / 16, Location.Y /
16).IsOpen Then
                        Open0Cells(g, GameCells(Location.X / 16, Location.Y /
16))
                    ' End If
                    Case "1"
                        g.DrawImage(My.Resources._1Cell, Location)
                    Case "2"
                        g.DrawImage(My.Resources._2Cell, Location)
                    Case "3"
                        g.DrawImage(My.Resources._3Cell, Location)
                    Case "4"
                        g.DrawImage(My.Resources._4Cell, Location)
                    Case "5"
                        g.DrawImage(My.Resources._5Cell, Location)
                    Case "6"
                        g.DrawImage(My.Resources._6Cell, Location)
                    Case "7"
                        g.DrawImage(My.Resources._7Cell, Location)
                    Case "8"
                        g.DrawImage(My.Resources._8Cell, Location)
                    Case "m"
                        If FirstMove Then ReDim
GameCells(GameCells.GetUpperBound(0), GameCells.GetUpperBound(1)) :
ResetGame(GameCells.GetUpperBound(0) + 1, GameCells.GetUpperBound(1) + 1) :
OpenCell(Location) : Exit Select
                        EndGameLost(Location, g)
                End Select
                If Ended <> "Loss" Then CheckIfWon(g)
            End If
            FirstMove = False
        End Using
    End Sub
```

```
    Sub CheckIfWon(g As Graphics)
        For Each Cell In GameCells
            If Cell.IsOpen Then OpenedCount += 1
        Next
        If OpenedCount = Board.Width / 16 * Board.Height / 16 - MineCount Then
            For Each Cell In GameCells
                If Cell.GetState = "m" Then g.DrawImage(My.Resources.FlagCell,
Cell.GetLocation)
            Next
            Ended = "Won"
        End If
        OpenedCount = 0
    End Sub

    Sub EndGameLost(Location As Point, g As Graphics)
        For Each Cell In GameCells
            If Cell.GetState = "m" Then g.DrawImage(My.Resources.LostMineCell,
Cell.GetLocation)
        Next
        g.DrawImage(My.Resources.OpenedMineCell, Location)
        Ended = "Loss"
    End Sub
    Sub Open0Cells(g As Graphics, Cell As GameCell)
        For Each Adjacent In Cell.GetAdjacents
            If Not Adjacent.IsOpen Then OpenCell(Adjacent.GetLocation)
        Next
    End Sub

    Sub GenerateMines()
        Dim r As New Random
        For m = 0 To MineCount - 1
            Do
                Dim x As Integer = r.Next(0, GameCells.GetUpperBound(0) + 1)
                Dim y As Integer = r.Next(0, GameCells.GetUpperBound(1) + 1)
                If GameCells(x, y).GetState <> "m" Then GameCells(x, y).GetState
= "m" : Exit Do
            Loop
        Next
    End Sub
    ReadOnly Property GetBoard As Bitmap
        Get
            Return Board
        End Get
    End Property

    ReadOnly Property GetMines As Integer
        Get
            Return MineCount
        End Get
    End Property

    Property IsFirstMove As Boolean
        Get
            Return FirstMove
        End Get
        Set(value As Boolean)
            FirstMove = value
        End Set
    End Property

    ReadOnly Property GetEndResult As String
        Get
```

```
            Return Ended
        End Get
    End Property

    ReadOnly Property GetCells() As GameCell(,)
        Get
            Return GameCells
        End Get
    End Property

    ReadOnly Property Get3BV As Integer
        Get
            Return _3BV
        End Get
    End Property

    ReadOnly Property GetDifficulty As String
        Get
            Return Difficulty
        End Get
    End Property
End Class

<Serializable> Public Class GameCell
    Private State As Char
    Private Index As Point
    Private Location As Point
    Private AdjacentCells As New List(Of GameCell)
    Private Opened As Boolean
    Private Flagged As Boolean
    Private Marked As Boolean
    Sub New(BoardIndex As Point)
        Index = BoardIndex
        Location = New Point(Index.X * 16, Index.Y * 16)
        State = ""
        Opened = False
        Marked = False
    End Sub

    Sub GenerateAdjacentCells(CellStates As GameCell(,), Dimensions As Point)
        Dim MineCount As Integer
        For x = -1 To 1
            For y = -1 To 1
                If x = 0 AndAlso y = 0 Then Continue For
                If Index.X + x <= Dimensions.X AndAlso Index.Y + y <=
Dimensions.Y AndAlso Index.X + x >= 0 AndAlso Index.Y + y >= 0 Then
                    If CellStates(Index.X + x, Index.Y + y).GetState = "m" Then
MineCount += 1
                    AdjacentCells.Add(CellStates(Index.X + x, Index.Y + y))
                End If
            Next
        Next
        CellStates(Index.X, Index.Y).GetState = CStr(MineCount)
    End Sub

    Property IsOpen As Boolean
        Get
            Return Opened
        End Get
        Set(value As Boolean)
            Opened = value
        End Set
    End Property
```

```
    Property GetState As Char
        Get
            Return State
        End Get
        Set(value As Char)
            State = value
        End Set
    End Property

    Property IsFlagged As Boolean
        Get
            Return Flagged
        End Get
        Set(value As Boolean)
            Flagged = value
        End Set
    End Property

    ReadOnly Property GetIndex As Point
        Get
            Return Index
        End Get
    End Property

    ReadOnly Property GetLocation As Point
        Get
            Return Location
        End Get
    End Property

    ReadOnly Property GetAdjacents As List(Of GameCell)
        Get
            Return AdjacentCells
        End Get
    End Property

    Property IsMarked As Boolean
        Get
            Return Marked
        End Get
        Set(value As Boolean)
            Marked = value
        End Set
    End Property

End Class
```

GameplayModule

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

Module GamePlayModule
    Public myMenuForm As MenuForm
    Public myStatsForm As OverallStatsForm
    Public myGame As New Game
    Public myGameplayForm As GameplayForm
    Public myLiveStatsForm As LiveStatsForm
    Public myTrainerForm As TrainerForm
    Public myCustomGameForm As CustomBoardForm
```

```
    Sub Main()
        ' startup for program, runs main menu form
        myMenuForm = New MenuForm
        myGameplayForm = New GameplayForm

        Application.Run(myMenuForm)
        Main()
    End Sub
    Function DeepClone(Of T)(ByRef orig As T) As T
        'used to create copies of data structures such that the original isnt
modified when the copy is
        If orig Is Nothing Then Return Nothing

        Dim formatter As New BinaryFormatter()
        Dim stream As New MemoryStream()

        formatter.Serialize(stream, orig)
        stream.Seek(0, SeekOrigin.Begin)

        Return formatter.Deserialize(stream)

    End Function
    Sub ErrorMessage(Message As String)
        MsgBox(Message, MsgBoxStyle.Critical, "Error")
    End Sub
    Class Game
        Private myConstraintSolver As ConstraintSolver
        Public GameOver As String = ""
        Public FirstMove As Boolean
        Private BoardStates As Char(,)
        Public Cells(,) As Cell
        Protected SafeCells As New List(Of Cell)
        Private CurrentStrategy As String
        Protected Guess As New BestGuessCell
        Public HardestAlgorithm As String

        Public Sub PlayGame()

            BoardStates = myGameplayForm.GetBoardStates


            Guess = New BestGuessCell
            FirstMove = True
            OpenStatsForm() : Application.DoEvents()

            Do Until myGameplayForm.Stopped ' main loop for gameplay

                BoardStates = myGameplayForm.GetBoardStates

                If BoardStates Is Nothing Then EndGame()

                If FirstMove Then myGameplayForm.ResetGame() : FirstMove = False
: myGameplayForm.OpenCellProgram(New Point(0, 0)) : GameOver = "" : Continue Do
                ' if its the first move then reset game and click on the top left
corner

                Cells = GetCells()
                If GameOver <> "" Then EndGame() : Continue Do

                SafeCells.Clear()
```

```
                If Guess.GetUsed Then myLiveStatsForm.GuessUpdate("Success")
                Guess = New BestGuessCell

                EvaluateBoard()

                If SafeCells.Count <> 0 Then

                    For Each SafeCell In SafeCells
                        myGameplayForm.OpenCellProgram(SafeCell.GetLocation)
                        CurrentStrategy = "Safe"
                    Next

                Else

                    If myConstraintSolver.GetGuess IsNot Nothing Then Guess =
myConstraintSolver.GetGuess
                    MakeInformedMove()

                End If
            Loop
        End Sub

        Private Sub OpenStatsForm()

            If myLiveStatsForm Is Nothing Then myLiveStatsForm = New
LiveStatsForm Else myLiveStatsForm.Show() : Exit Sub

            Dim r As New Random

            If Not myGameplayForm.Bounds.IntersectsWith(myLiveStatsForm.Bounds)
Then

                myLiveStatsForm.Show() 'display live stats form if it doesnt
overlap with the game board

            Else

                If myGameplayForm.Bounds.Width + myLiveStatsForm.Bounds.Width >
Screen.PrimaryScreen.Bounds.Width OrElse myGameplayForm.Bounds.Height +
myLiveStatsForm.Bounds.Height > Screen.PrimaryScreen.Bounds.Height Then Exit Sub
                'if the live stats form is too big to fit on the screen with no
overlap then dont display it at all
                Do
                    myLiveStatsForm.Bounds = New
Rectangle(r.Next(Screen.PrimaryScreen.Bounds.Width - myLiveStatsForm.Width),
r.Next(Screen.PrimaryScreen.Bounds.Height - myLiveStatsForm.Height),
myLiveStatsForm.Bounds.Width, myLiveStatsForm.Bounds.Height)
                    'randomly assign the form a location until it doesnt
intersect with the board
                Loop Until Not
myGameplayForm.Bounds.IntersectsWith(myLiveStatsForm.Bounds)
                myLiveStatsForm.Show()
            End If
        End Sub
        Private Function EvaluateBoard()
            For Each Cell In Cells
                Cell.GenerateAdjacents(Cells, BoardStates.GetUpperBound(0),
BoardStates.GetUpperBound(1))
                Cell.CheckIfMine()
                Cell.CheckIfUseless()
            Next
            Dim SafeCount As Integer = -1
            SafeCount = SafeCells.Count
```

```
            For Each Cell In Cells ' running the simple algorithm on all relevant
cells
                For Each SafeCell In Cell.SimpleFinder()
                    If Not SafeCells.Contains(SafeCell) Then
SafeCells.Add(SafeCell)
                    If HardestAlgorithm = "" Then HardestAlgorithm = "Simple"
                Next
            Next
            If SafeCells.Count = 0 Then
                myConstraintSolver = New ConstraintSolver(Cells, SafeCells,
False) 'running the constraint solver
                myConstraintSolver.FindNonTrivialSolutions()
                SafeCells = myConstraintSolver.GetSafeCells
            End If
            Dim ReplacementSafeCells As New List(Of Cell) ' replacing cells from
copied data structure to original data structure
            For Each Cell In SafeCells
                    ReplacementSafeCells.Add(Cells(Cell.GetIndex.X,
Cell.GetIndex.Y))
                Next
                SafeCells = ReplacementSafeCells
                For Each Cell In Cells
                    If Cell.Adjacents.Except(SafeCells).Count =
Val(Cell.CellContents) Then
                        For Each Mine In Cell.Adjacents.Except(SafeCells)
                            Mine.IsMine = True
                        Next
                    End If
                Next
            Return SafeCells
        End Function
        Private Sub MakeInformedMove()
            HardestAlgorithm = "Guess"
            If Guess.GetCell IsNot Nothing Then
                myGameplayForm.OpenCellProgram(Guess.GetCell.GetLocation)
                Guess.GetUsed = True
                CurrentStrategy = "Guess"
            ElseIf FindCorner() IsNot Nothing Then
                myGameplayForm.OpenCellProgram(FindCorner.GetLocation)
            Else
                ClickRandom(False)
            End If
        End Sub

        Private Function FindCorner() As Cell
            'looks at each corner and returns the first unopened one
            If Cells(0, Cells.GetUpperBound(1)).CellContents = "u" Then
                CurrentStrategy = "Corner"
                Return Cells(0, Cells.GetUpperBound(1))
            ElseIf Cells(Cells.GetUpperBound(0), 0).CellContents = "u" Then
                CurrentStrategy = "Corner"
                Return Cells(Cells.GetUpperBound(0), 0)
            ElseIf Cells(Cells.GetUpperBound(0),
Cells.GetUpperBound(1)).CellContents = "u" Then
                CurrentStrategy = "Corner"
                Return Cells(Cells.GetUpperBound(0), Cells.GetUpperBound(1))
            End If
            Return Nothing
        End Function

        Function SafeToClick(Cell As Cell) As Boolean
            ' cells are regarded as safe if they have two "layers" of unknown
cells surrounding them
```

```
            If Cell.IsMine Then Return False
            Dim x = Cell.GetIndex.X
            Dim y = Cell.GetIndex.Y
            If x = 0 Then
                For a = 0 To 2
                    For b = 0 To -2
                        If Cells(x + a, y + b).Adjacents.Count <> 0 Then Return
False
                    Next
                Next
            ElseIf y = 0 Then
                For a = 0 To -2
                    For b = 0 To 2
                        If Cells(x + a, y + b).Adjacents.Count <> 0 Then Return
False
                    Next
                Next
            Else
                For a = 0 To -2
                    For b = 0 To -2
                        If Cells(x + a, y + b).Adjacents.Count <> 0 Then Return
False
                    Next
                Next
            End If
            Return True
        End Function
        Private Sub EndGame()
            myLiveStatsForm.GameEndUpdate(GameOver)
            If Guess.GetUsed And GameOver = "Loss" Then
myLiveStatsForm.GuessUpdate("Fail")
            If GameOver = "Loss" Then
                myLiveStatsForm.UpdateTable(CurrentStrategy)
            End If
            FirstMove = True
            myGameplayForm.ResetGame()
            HardestAlgorithm = ""
        End Sub
        Overridable Function GetCells() As Cell(,)
            ReDim Cells(BoardStates.GetUpperBound(0),
BoardStates.GetUpperBound(1))
            For x = 0 To BoardStates.GetUpperBound(0)
                For y = 0 To BoardStates.GetUpperBound(1)
                    Cells(x, y) = New Cell(BoardStates(x, y), New Point(x, y))
                Next
            Next
            Return Cells
        End Function
        Private Sub ClickRandom(Skip As Boolean)
            For Each Cell In Cells
                If Cell.CellContents = "u" And Not Cell.IsMine And (Skip Or
SafeToClick(Cell)) Then
                    myGameplayForm.OpenCellProgram(Cell.GetLocation)
                    CurrentStrategy = "Random"
                    Exit Sub
                End If
            Next
            ClickRandom(True) ' runs again if no cell is deemed "SafeToClick" and
resorts to one which isnt to avoid becoming stuck
        End Sub

    End Class
```

```
    Class ConstraintSolver
        Private Guess As New BestGuessCell
        Private CustomBoard As Boolean
        Private SafeCells As List(Of Cell)
        Private OldCells(,) As Cell
        Private Cells(,) As Cell
        Private EquationList As New List(Of CoupledEquation)
        Private VariableList As New List(Of Variable)
        Private AllSolutions As New List(Of List(Of Solution))
        Private SolutionCombos As New List(Of SolutionCombo)
        Private TotalPermutations As Double
        Private SolutionDict As New Dictionary(Of Integer, List(Of
SolutionCombo))
        Private MineCells As Integer
        Private UnknownCells As Integer
        Private ShortEval As Boolean

        Sub New(CellArray As Cell(,), SafeCellList As List(Of Cell), Custom As
Boolean)
            CustomBoard = Custom
            OldCells = CellArray
            Cells = DeepClone(CellArray) ' two copies needed so changing one wont
affect the other
            SafeCells = SafeCellList
        End Sub

        Public Sub FindNonTrivialSolutions()
            For Each Cell In Cells
                Cell.ReduceCell()
            Next
            Dim SharedAdjacentCount As Integer
            For Each Cell In Cells
                If Not Cell.IsUseless Then ' only consider relevant cells
                    For Each OtherCell In Cells
                        If Not OtherCell.IsUseless AndAlso Cell IsNot OtherCell
AndAlso Not Cell.Adjacents.Count < OtherCell.Adjacents.Count Then ' only if cell
has a higher number of adjacents then othercell to avoid negative errors
                            SharedAdjacentCount =
Cell.CountSharedAdjacents(OtherCell)
                            If SharedAdjacentCount = OtherCell.Adjacents.Count
Then ' if cell contains all the adjacents of other cell then other cell can be
substituted into it
                                Cell.RemoveSharedAdjacents(OtherCell)
                                If Cell.CellContents = "-" Then SafeCells =
Nothing : Exit Sub ' only relevant to custom boards, detects if board is invalid
                                If Val(Cell.CellContents) = 0 Then ' if there are
no more mines left after the subtraction then the remaining adjacents must be
safe
                                    For Each Adjacent In Cell.Adjacents
                                        If SafeCells.Find(Function(x) x.GetIndex
= Adjacent.GetIndex) Is Nothing AndAlso Not Cells(OtherCell.GetIndex.X,
OtherCell.GetIndex.Y).Adjacents.Contains(Adjacent) Then
                                            SafeCells.Add(Adjacent)
                                        End If
                                    Next
                                    OldCells(OtherCell.GetIndex.X,
OtherCell.GetIndex.Y).UpdateMines(SafeCells)
                                    Cell.IsUseless = True
                                End If
                            End If
                        End If
                    Next
                End If
```

```
            Next
            For Each Cell In Cells ' more updating of mines
                If Cell.Adjacents.Count = Val(Cell.CellContents) And
Cell.Adjacents.Count <> 0 Then
                    For Each Mine In Cell.Adjacents
                        If Mine.IsMine = False Then Mine.IsMine = True
                        If Not OldCells(Mine.GetIndex.X, Mine.GetIndex.Y).IsMine
Then OldCells(Mine.GetIndex.X, Mine.GetIndex.Y).IsMine = True
                    Next
                ElseIf Cell.Adjacents.Count < Val(Cell.CellContents) Then '
custom board validity check
                    SafeCells = Nothing
                    Exit Sub
                End If
            Next
            For Each Cell In OldCells ' simple finder to check for any newly
found safe cells
                For Each SafeCell In Cell.SimpleFinder
                    If Not SafeCells.Contains(SafeCell) Then
SafeCells.Add(SafeCell)
                Next
            Next
            If SafeCells.Count = 0 Then
                For Each Cell In Cells
                    Cell.ReduceCell()
                    If Cell.IsUseless = False Then AddToEquation(Cell)
                Next
                CombineEquations() ' combine equations with common variables to
save useless computation

                EvaluateEquationsLong() 'move to brute force if no safe cells are
found
            ElseIf Not CustomBoard Then
                If myGame IsNot Nothing Then If myGame.HardestAlgorithm = "Safe"
OrElse myGame.HardestAlgorithm = "" Then myGame.HardestAlgorithm = "Non-Trivial"
            End If
        End Sub

        Private Sub AddToEquation(Cell As Cell)
            Dim AddedtoEquation As Boolean = False
            If EquationList.Count = 0 Then
                EquationList.Add(New CoupledEquation(Cell, EquationList,
VariableList))
            Else
                For Each Equation In EquationList
                    For Each Adjacent In Cell.Adjacents
                        If Equation.GetVariables.Count + Cell.Adjacents.Count <
20 Then ' more than 20 variables in an equation becomes slow to compute
                            If Equation.GetVariables.Find(Function(x) x.GetCell
Is Adjacent) IsNot Nothing Then ' if one of the cells adjacents is in an equation
already then add to that one
                                Equation.AddVariable(Cell, EquationList,
VariableList)
                                AddedtoEquation = True
                                Exit For
                            End If
                        Else
                            ShortEval = True
                        End If
                    Next
                    If AddedtoEquation Then Exit For
                Next
```

```
                If Not AddedtoEquation Then EquationList.Add(New
CoupledEquation(Cell, EquationList, VariableList)) ' if no equations contain any
of the cells adjacents then make a new one
            End If
        End Sub

        Private Sub CombineEquations()
            For Each Equation In EquationList
                Dim VariableReplacements As New List(Of Variable)
                For Each Variable In Equation.GetVariables ' finding all copies
of variables in equations which aren't the same in memory
                    If Not VariableList.Contains(Variable) Then
                        VariableReplacements.Add(VariableList.Find(Function(x)
x.GetCell Is Variable.GetCell))
                    End If
                Next
                For Each Variable In VariableReplacements ' replacing the bad
copies with a common version of each so they're updated simultaneously

Equation.GetVariables(Equation.GetVariables.FindIndex(Function(x) x.GetCell Is
Variable.GetCell)) = Variable
                    For Each Constraint In Equation.GetConstraints ' doing the
same for constraints
                        If Constraint.GetVariables.Contains(Variable) Then
Constraint.GetVariables(Constraint.GetVariables.FindIndex(Function(x) x.GetCell
Is Variable.GetCell)) = Variable
                    Next
                Next
            Next
            Dim EquationsToRemove As New List(Of CoupledEquation)
            For Each Equation In EquationList
                For Each OtherEquation In EquationList
                    If Equation IsNot OtherEquation AndAlso Not
EquationsToRemove.Contains(OtherEquation) AndAlso Equation.GetVariables.Count >=
OtherEquation.GetVariables.Count Then ' avoiding negative errors when comparing
equations
                        If Equation.GetVariables.Count +
OtherEquation.GetVariables.Except(Equation.GetVariables).Count <= 20 Then '
keeping variable count under 20
                            If Equation.GetVariables.Find(Function(x)
OtherEquation.GetVariables.Contains(x)) IsNot Nothing Then ' if they share a
variable then combine them
                                For Each Variable In OtherEquation.GetVariables
                                    If Not
Equation.GetVariables.Contains(Variable) Then Equation.GetVariables.Add(Variable)
                                Next
                                For Each Constraint In
OtherEquation.GetConstraints
                                    If Not
Equation.GetConstraints.Contains(Constraint) Then
Equation.GetConstraints.Add(Constraint)
                                Next
                                EquationsToRemove.Add(OtherEquation)
                            End If
                        Else
                            If
OtherEquation.GetVariables.Except(Equation.GetVariables) Is Nothing Then ' if
equation has all the variables other equation contains then remove it anyway
                                EquationsToRemove.Add(OtherEquation)
                            Else
                                ShortEval = True
                            End If
                        End If
```

```
                End If
            Next
        Next
        For Each Equation In EquationsToRemove ' remove excess equations
            EquationList.Remove(Equation)
        Next
    End Sub


    Function EvaluateEquationsLong() As Boolean
        Dim Solver As BruteForceSolver
        Dim TotalCombos As Long = 1
        myGame.HardestAlgorithm = "Brute Force"
        For Each Cell In Cells
            If Cell.IsMine Then MineCells += 1
            If Cell.CellContents = "u" And Cell.IsMine = False Then
UnknownCells += 1
        Next
        If Not CustomBoard Then MineCells = myGameplayForm.GetMines -
MineCells
        For Each Equation In EquationList ' finding solutions for each
equation
            Solver = New BruteForceSolver(Equation, Cells, SafeCells,
CustomBoard)
            Solver.FindSolutions()
            Solver.DeduceSafeCells()
            If SafeCells.Count <> 0 Then Return False
            AllSolutions.Add(Solver.GetSolutions)
            TotalCombos *= Solver.GetSolutions.Count
        Next
        If TotalCombos > 500000 Or ShortEval Then Guess.ShortEval(Cells) :
Return False ' more than 500000 would take too long to evaluate
        Dim Count As New List(Of Integer)
        For x = 0 To AllSolutions.Count - 1
            Count.Add(0)
        Next
        EnumeratePermutations(Count) 'takes a while, finding all combinations
of solutions
        For Each Combo In SolutionCombos ' categorising solution combos by
number of mines
            Combo.CountMines()
            If Not SolutionDict.ContainsKey(Combo.GetMines) Then
                SolutionDict.Add(Combo.GetMines, New List(Of SolutionCombo))
                Combo.FindPermutations(OldCells, MineCells, UnknownCells,
CustomBoard)
                SolutionDict(Combo.GetMines).Add(Combo)
            Else
                Combo.GetPermutations(OldCells) =
SolutionDict(Combo.GetMines)(0).GetPermutations(OldCells) 'takes a long time
                Combo.MinesLeft = SolutionDict(Combo.GetMines)(0).MinesLeft
                Combo.CellsLeft = SolutionDict(Combo.GetMines)(0).CellsLeft
            End If
            TotalPermutations += Combo.GetPermutations(OldCells)
        Next
        For Each Cell In OldCells
            If Cell.GetOccurences <> 0 Then
                Cell.GetChanceSafe = 1 - (Cell.GetOccurences /
TotalPermutations)
                If Cell.GetChanceSafe = 0 Then Cell.IsMine = True
                Guess.BestGuess(Cell, Cell.GetChanceSafe)
            End If
        Next
```

```
            If Not CustomBoard Then RandomCellChance()
            For Each Cell In OldCells
                Cell.GetOccurences = 0
            Next
            Return True
        End Function

        Sub RandomCellChance()
            ' finding the chance that a completely unkown cell is safe
            Dim Chance As Double
            Dim SolutionTotal As Double
            For Each S In SolutionDict
                Chance += S.Value(0).MinesLeft / S.Value(0).CellsLeft *
S.Value(0).GetPermutations(OldCells) * S.Value.Count
                SolutionTotal += S.Value(0).GetPermutations(OldCells) *
S.Value.Count
            Next
            Chance = 1 - (Chance / SolutionTotal)
            If Chance > Guess.GetChanceSafe Then Guess.RandomCell(Chance)
        End Sub

        Private Sub EnumeratePermutations(Count As List(Of Integer))
            Dim blnFinished As Boolean
            Do Until blnFinished
                WritePerm(Count)
                blnFinished = GetNext(Count)
            Loop
        End Sub

        Private Sub WritePerm(Count As List(Of Integer))
            Dim NewCombo As New List(Of Solution)
            For i As Integer = 0 To AllSolutions.Count - 1
                NewCombo.Add(AllSolutions(i)(Count(i)))
            Next
            SolutionCombos.Add(New SolutionCombo(NewCombo))
        End Sub

        Private Function GetNext(Count As List(Of Integer)) As Boolean
            For i As Integer = AllSolutions.Count - 1 To 0 Step -1
                If Count(i) < AllSolutions(i).Count - 1 Then
                    Count(i) += 1
                    Return False
                Else
                    Count(i) = 0
                End If
            Next
            Return True
        End Function

        ReadOnly Property GetSafeCells As List(Of Cell)
            Get
                Return SafeCells
            End Get
        End Property

        ReadOnly Property GetGuess As BestGuessCell
            Get
                Return Guess
            End Get
        End Property

    End Class
```

```
    Class BruteForceSolver
        Private CustomBoard As Boolean
        Private Cells(,) As Cell
        Private SafeCells As List(Of Cell)
        Private Equation As CoupledEquation
        Private VariableStack As New Stack(Of Variable)
        Private ReturnToStack As New Stack(Of Variable)
        Private VariablesToCheck As New List(Of Variable)
        Private Solutions As New List(Of Solution)
        Private SolutionFound As Boolean = True
        Public Sub New(EquationToSolve As CoupledEquation, CellArray(,) As Cell,
SafeCellList As List(Of Cell), Custom As Boolean)
            CustomBoard = Custom
            Cells = CellArray
            SafeCells = SafeCellList
            Equation = EquationToSolve
            Equation.GetVariables =
Equation.GetVariables.OrderByDescending(Function(x) x.GetTally).ToList 'ordering
variables by number of occurences
            VariablesToCheck = Equation.GetVariables
        End Sub

        Public Sub FindSolutions()
            Do
                CreateStack()
                For Each Constraint In Equation.GetConstraints
                    If Not Constraint.IsSatisfied Then
                        SolutionFound = False
                        Exit For
                    End If
                Next
                If SolutionFound Then
                    Solutions.Add(New Solution(DeepClone(VariableStack).ToList))
                End If
                SolutionFound = True
                If ReturnToStack.Count = 0 Then Exit Do
                Do Until VariableStack.Peek Is ReturnToStack.Peek
                    VariableStack.Pop()
                Loop
                ReturnToStack.Pop()
                VariablesToCheck =
Equation.GetVariables.GetRange(Equation.GetVariables.IndexOf(VariableStack.Peek),
Equation.GetVariables.Count - Equation.GetVariables.IndexOf(VariableStack.Pop)) '
finding the variables which are not in the stack
                For Each Variable In VariablesToCheck
                    If Variable IsNot VariablesToCheck(0) Then Variable.GetStatus
= 0
                Next
            Loop
        End Sub

        Private Sub CreateStack()
            For Each Variable In VariablesToCheck
                If Variable.GetStatus = 1 Then
                    Variable.GetStatus = 0
                ElseIf Variable.GetStatus = 0 Then
                    Variable.GetStatus = 1
                End If
                For Each Constraint In Equation.GetConstraints
                    If Constraint.GetVariables.Find(Function(x)
x.GetCell.GetIndex = Variable.GetCell.GetIndex) IsNot Nothing Then
```

```
                        If GetMineCount(VariablesToCheck.FindAll(Function(x)
Constraint.GetIndexes.Contains(x.GetCell.GetIndex))) > Constraint.GetTotalMines
Then
                            Variable.GetStatus = 0
                        End If
                    End If
                Next
                If Variable.GetStatus = 1 Then ReturnToStack.Push(Variable)
                VariableStack.Push(Variable)
            Next
        End Sub

        Private Function GetMineCount(ByRef Variables As List(Of Variable)) As
Integer
            ' number of mines in the list of variables
            Return TryCast(Variables, IEnumerable(Of Variable)).Count(Function(x)
x.GetStatus = 1)
        End Function

        Public Sub DeduceSafeCells()
            Dim CheckForConsistency As New List(Of Variable)
            Dim CurrentCell As Cell
            If CustomBoard OrElse Solutions.Count = 0 Then Exit Sub
            For x = 0 To Solutions(0).GetVariables.Count - 1
                For Each Solution In Solutions

CheckForConsistency.Add(Solution.GetVariables.Find(Function(y) y.GetCell.GetIndex
= Solutions(0).GetVariables(x).GetCell.GetIndex)) ' adding each instance of a
variable in solutions to a list
                Next
                CurrentCell = Cells(CheckForConsistency(0).GetCell.GetIndex.X,
CheckForConsistency(0).GetCell.GetIndex.Y)
                CurrentCell.GetChanceSafe = 1 - (TryCast(CheckForConsistency,
IEnumerable(Of Variable)).Count(Function(y) y.GetStatus = 1) / Solutions.Count)
                If CurrentCell.GetChanceSafe = 0 Then CurrentCell.IsMine = True
                If CurrentCell.GetChanceSafe = 1 Then SafeCells.Add(CurrentCell)
                CheckForConsistency.Clear()
            Next

        End Sub

        ReadOnly Property GetSolutions As List(Of Solution)
            Get
                Return Solutions
            End Get
        End Property
    End Class

    Class SolutionCombo
        Private Solutions As List(Of Solution)
        Private NumberOfOccurences As Double
        Private NumberOfMines As Integer
        Private MinesLeftOnBoard As Integer
        Private CellsLeftOnBoard As Integer

        Sub New(SolutionList As List(Of Solution))
            Solutions = SolutionList
        End Sub

        Sub CountMines()
            For Each Solution In Solutions
                NumberOfMines += Solution.GetMines
            Next
```

```
        End Sub

        Sub FindPermutations(OldCells As Cell(,), MineCells As Integer,
UnknownCells As Integer, CustomBoard As Boolean)
            ' using the choose function to find permutations
            If Not CustomBoard Then MinesLeftOnBoard = MineCells - NumberOfMines
Else MinesLeftOnBoard = NumberOfMines
            If MinesLeftOnBoard >= 0 Then
                Dim NumberCells As Integer
                For Each Solution In Solutions
                    NumberCells += Solution.GetVariables.Count
                Next
                CellsLeftOnBoard = UnknownCells - NumberCells
                Dim r As Integer = MinesLeftOnBoard
                If r > CellsLeftOnBoard / 2 Then r = CellsLeftOnBoard - r
                NumberOfOccurences = Pascal(CellsLeftOnBoard, r)
                For Each Solution In Solutions
                    Solution.AddOccurences(NumberOfOccurences, OldCells)
                Next
            Else
                NumberOfOccurences = 0
            End If
        End Sub

        Function Pascal(n As Integer, r As Integer)
            Dim ans As String = "1"
            For x As Integer = 0 To r - 1
                ans = Math.Round(ans * ((n - x) / (x + 1)))
            Next
            Return ans
        End Function

        Property GetPermutations(OldCells As Cell(,)) As Double
            Get
                Return NumberOfOccurences
            End Get
            Set(value As Double)
                NumberOfOccurences = value
                For Each Solution In Solutions
                    Solution.AddOccurences(NumberOfOccurences, OldCells)
                Next
            End Set
        End Property

        Property CellsLeft As Integer
            Get
                Return CellsLeftOnBoard
            End Get
            Set(value As Integer)
                CellsLeftOnBoard = value
            End Set
        End Property

        Property MinesLeft As Integer
            Get
                Return MinesLeftOnBoard
            End Get
            Set(value As Integer)
                MinesLeftOnBoard = value
            End Set
        End Property

        ReadOnly Property GetMines As Integer
```

```
        Get
            Return NumberOfMines
        End Get
    End Property
End Class

Class BestGuessCell
    Private Cell As Cell
    Private ChanceSafe As Decimal
    Private Used As Boolean
    Public Sub BestGuess(NewGuess As Cell, NewChance As Decimal)
        If Cell Is Nothing Then
            Cell = NewGuess
            ChanceSafe = NewChance
        ElseIf NewChance > ChanceSafe Then
            Cell = NewGuess
            ChanceSafe = NewChance
        End If
    End Sub

    Sub RandomCell(Chance As Decimal)
        Cell = Nothing
        ChanceSafe = Chance
    End Sub

    Sub ShortEval(Cells As Cell(,))
        For Each C In Cells
            If C.GetChanceSafe > ChanceSafe Then
                Cell = C
                ChanceSafe = C.GetChanceSafe
            End If
        Next
    End Sub

    ReadOnly Property GetCell As Cell
        Get
            Return Cell
        End Get
    End Property

    ReadOnly Property GetChanceSafe As Decimal
        Get
            Return ChanceSafe
        End Get
    End Property

    Property GetUsed As Boolean
        Get
            Return Used
        End Get
        Set(value As Boolean)
            Used = value
        End Set
    End Property
End Class

<Serializable> Class Solution
    Private Variables As List(Of Variable)
    Private Mines As Integer

    Public Sub New(VariablesinSolution As List(Of Variable))
        Variables = VariablesinSolution
        Mines = Variables.FindAll(Function(x) x.GetStatus = 1).Count
```

```
        End Sub

        Sub AddOccurences(NumOfOccurences As Double, OldCells As Cell(,))
            For Each Variable In Variables
                OldCells(Variable.GetCell.GetIndex.X,
Variable.GetCell.GetIndex.Y).GetOccurences += Variable.GetStatus *
NumOfOccurences
            Next
        End Sub

        ReadOnly Property GetVariables As List(Of Variable)
            Get
                Return Variables
            End Get
        End Property

        ReadOnly Property GetMines As Integer
            Get
                Return Mines
            End Get
        End Property

    End Class

    <Serializable> Class Constraint
        Private Variables As New List(Of Variable)
        Private Mines As Integer

        Public Sub New(Cell As Cell, NewVariables As List(Of Variable), ByRef
EquationList As List(Of CoupledEquation))
            For Each Equation In EquationList
                If Equation.GetConstraints.Find(Function(x) x.GetVariables Is
NewVariables And x.GetTotalMines = Val(Cell.CellContents)) IsNot Nothing Then
Exit Sub
            Next
            Variables.AddRange(NewVariables)
            Mines = Val(Cell.CellContents)
        End Sub


        ReadOnly Property IsSatisfied As Boolean
            Get
                If TryCast(Variables, IEnumerable(Of Variable)).Count(Function(x)
x.GetStatus = 1) = Mines Then Return True
                Return False
            End Get
        End Property
        ReadOnly Property GetVariables As List(Of Variable)
            Get
                Return Variables
            End Get
        End Property
        ReadOnly Property GetTotalMines As Integer
            Get
                Return Mines
            End Get
        End Property

        ReadOnly Property GetIndexes As List(Of Point)
            Get
                Dim Indexes As New List(Of Point)
                For Each Variable In Variables
                    Indexes.Add(Variable.GetCell.GetIndex)
```

```
                Next
                Return Indexes
            End Get
        End Property

        ReadOnly Property GetStatuses As List(Of Integer)
            Get
                Dim Status As New List(Of Integer)
                For Each Variable In Variables
                    Status.Add(Variable.GetStatus)
                Next
                Return Status
            End Get
        End Property
    End Class

    <Serializable> Class Variable
        Private Cell As Cell
        Private Tally As Integer
        Private Status As Integer
        Public Sub New(NewCell As Cell)
            Cell = NewCell
            Tally = 0
            Status = 0
        End Sub

        ReadOnly Property GetCell As Cell
            Get
                Return Cell
            End Get
        End Property
        Property GetTally As Integer
            Get
                Return Tally
            End Get
            Set(value As Integer)
                Tally = value
            End Set
        End Property
        Property GetStatus As Integer
            Get
                Return Status
            End Get
            Set(value As Integer)
                Status = value
            End Set
        End Property
    End Class
    <Serializable> Class CoupledEquation
        Private Variables As New List(Of Variable)
        Private Constraints As New List(Of Constraint)
        Public Sub New(Cell As Cell, ByRef EquationList As List(Of
CoupledEquation), ByRef VariableList As List(Of Variable))
            For Each Adjacent In Cell.Adjacents
                Variables.Add(New Variable(Adjacent))
            Next
            For Each Variable In Variables
                If Not VariableList.Contains(Variable) Then
VariableList.Add(Variable)
            Next
            Constraints.Add(New Constraint(Cell, VariableList.FindAll(Function(x)
Cell.Adjacents.Contains(x.GetCell)), EquationList))
        End Sub
```

```
        Public Sub AddVariable(Cell As Cell, ByRef EquationList As List(Of
CoupledEquation), ByRef VariableList As List(Of Variable))
            Dim Found = False
            For Each Adjacent In Cell.Adjacents
                For Each Variable In Variables
                    If Adjacent Is Variable.GetCell Then : Found = True : Exit
For : End If
                Next
                If Not Found Then Variables.Add(New Variable(Adjacent))
                Found = False
            Next
            For Each Variable In Variables
                If VariableList.Find(Function(x) x.GetCell Is Variable.GetCell)
Is Nothing Then VariableList.Add(Variable)
            Next
            Constraints.Add(New Constraint(Cell, VariableList.FindAll(Function(x)
Cell.Adjacents.Contains(x.GetCell)), EquationList))
        End Sub
        Property GetVariables() As List(Of Variable)
            Get
                Return Variables
            End Get
            Set(value As List(Of Variable))
                Variables = value
            End Set
        End Property

        ReadOnly Property GetConstraints As List(Of Constraint)
            Get
                Return Constraints
            End Get
        End Property

    End Class

    <Serializable> Class Cell
        Private Colour As String
        Private Contents As Char
        Private Location As Point
        Private Index As Point
        Private AdjacentCells As New List(Of Cell)
        Private Useless As Boolean
        Private Mine As Boolean
        Private Occurences As Double = 0
        Private ChanceSafe As Decimal


        Public Sub New(Board As Bitmap, CellLocation As Point, CellIndex As
Point)
            Mine = False
            Useless = False
            Location = CellLocation
            Index = CellIndex
            GetCellColour(Board)
            Contents = GetCellState()
        End Sub

        Public Sub New(State As Char, ArrIndex As Point)
            Contents = State
            Index = ArrIndex
            Location = New Point(ArrIndex.X * 16, ArrIndex.Y * 16)
            Mine = False
```

```
            Useless = False
        End Sub

        Private Sub GetCellColour(Board As Bitmap)
            Colour = ColorTranslator.ToHtml(Board.GetPixel(Location.X,
Location.Y))
            If Colour = "#C0C0C0" Then
                If ColorTranslator.ToHtml(Board.GetPixel(Location.X + 1,
Location.Y)) = "#000000" Then
                    Colour = "#000000"
                ElseIf ColorTranslator.ToHtml(Board.GetPixel(Location.X - 7,
Location.Y - 7)) <> "#FFFFFF" Then
                    Colour = "#FFFFFF"
                End If
            ElseIf Colour = "#000000" Then
                If ColorTranslator.ToHtml(Board.GetPixel(Location.X, Location.Y -
1)) = "#FF0000" Then
                    Colour = "#C0C0C0"
                    Mine = True
                Else
                    Colour = Nothing
                End If
            End If
        End Sub
        Private Function GetCellState() As String
            If Colour = "#C0C0C0" Then Useless = True : Return "u"
            If Colour = "#FFFFFF" Then Useless = True : Return "0"
            If Colour = "#0000FF" Then Return "1"
            If Colour = "#008000" Then Return "2"
            If Colour = "#FF0000" Then Return "3"
            If Colour = "#000080" Then Return "4"
            If Colour = "#800000" Then Return "5"
            If Colour = "#008080" Then Return "6"
            If Colour = "#000000" Then Return "7"
            If Colour = "#808080" Then Return "8"
            Return Nothing
        End Function

        Public Sub CheckIfUseless()
            Dim MineCount = 0
            If AdjacentCells.Count <> 0 And Not Useless Then
                For Each AdjacentCell In AdjacentCells
                    If AdjacentCell.Mine Then MineCount += 1
                Next
                If MineCount = AdjacentCells.Count Then Useless = True
            Else
                Useless = True
            End If
        End Sub

        Public Sub GenerateAdjacents(Cells(,) As Cell, BoardWidth As Integer,
BoardHeight As Integer)
            If Contents <> "u" Then
                AdjacentCells.Clear()
                If Index.X <> BoardWidth Then If Cells(Index.X + 1,
Index.Y).CellContents = "u" Then AdjacentCells.Add(Cells(Index.X + 1, Index.Y))
                If Index.Y <> BoardHeight Then If Cells(Index.X, Index.Y +
1).CellContents = "u" Then AdjacentCells.Add(Cells(Index.X, Index.Y + 1))
                If Index.X <> 0 Then If Cells(Index.X - 1,
Index.Y).CellContents = "u" Then AdjacentCells.Add(Cells(Index.X - 1, Index.Y))
                If Index.Y <> 0 Then If Cells(Index.X, Index.Y -
1).CellContents = "u" Then AdjacentCells.Add(Cells(Index.X, Index.Y - 1))
```

```
                    If Index.X <> BoardWidth And Index.Y <> BoardHeight Then If
Cells(Index.X + 1, Index.Y + 1).CellContents = "u" Then
AdjacentCells.Add(Cells(Index.X + 1, Index.Y + 1))
                    If Index.X <> 0 And Index.Y <> 0 Then If Cells(Index.X - 1,
Index.Y - 1).CellContents = "u" Then AdjacentCells.Add(Cells(Index.X - 1, Index.Y
- 1))
                    If Index.X <> BoardWidth And Index.Y <> 0 Then If
Cells(Index.X + 1, Index.Y - 1).CellContents = "u" Then
AdjacentCells.Add(Cells(Index.X + 1, Index.Y - 1))
                    If Index.X <> 0 And Index.Y <> BoardHeight Then If
Cells(Index.X - 1, Index.Y + 1).CellContents = "u" Then
AdjacentCells.Add(Cells(Index.X - 1, Index.Y + 1))
                End If
        End Sub

        Public Sub CheckIfMine()
            If Not Useless Then
                If Adjacents.Count = Val(Contents) Then
                    For Each Cell In AdjacentCells
                        Cell.Mine = True
                    Next
                End If
            End If
        End Sub



        Public Function SimpleFinder() As List(Of Cell)
            Dim SafeCells As New List(Of Cell)
            If Contents <> "u" Then
                Dim MineCount = 0
                For Each Cell In AdjacentCells
                    If Cell.Mine Then MineCount += 1
                Next
                If MineCount = Val(Contents) Then
                    For Each Cell In AdjacentCells
                        If Not Cell.Mine Then
                            SafeCells.Add(Cell)
                            Useless = True
                        End If
                    Next
                ElseIf MineCount > Val(Contents) Then
                    SafeCells = Nothing
                End If
            End If
            Return SafeCells
        End Function

        Public Sub ReduceCell()
            If Val(Contents) > AdjacentCells.Count Then Useless = True
            If Useless = False Then
                Dim CellsToRemove As New List(Of Cell)
                For Each Cell In AdjacentCells
                    If Cell.Mine = True Then
                        CellsToRemove.Add(Cell)
                        Contents = CStr(Val(Contents) - 1)
                    End If
                Next
                For Each Cell In CellsToRemove
                    AdjacentCells.Remove(Cell)
                Next
            End If
        End Sub
```

```
        Public Function CountSharedAdjacents(OtherCell As Cell) As Integer
            Dim Count As Integer = 0
            For Each AdjacentCell In OtherCell.Adjacents
                If Adjacents.Contains(AdjacentCell) Then
                    Count += 1
                End If
            Next
            Return Count
        End Function

        Public Sub RemoveSharedAdjacents(OtherCell As Cell)
            For Each AdjacentCell In OtherCell.Adjacents
                Adjacents.Remove(AdjacentCell)
            Next
            Contents = CStr(Val(Contents) - Val(OtherCell.Contents))
        End Sub

        Public Sub UpdateMines(SafeCells As List(Of Cell))
            Dim SafeCount As Integer = 0
            For Each AdjacentCell In Adjacents
                If SafeCells.Find(Function(x) x.Index = AdjacentCell.Index) IsNot
Nothing Then SafeCount += 1
            Next
            If Adjacents.Count - SafeCount = Val(Contents) Then
                For Each AdjacentCell In Adjacents
                    If SafeCells.Find(Function(x) x.Index = AdjacentCell.Index)
Is Nothing Then AdjacentCell.Mine = True
                Next
            End If
        End Sub

        Property GetOccurences As Double
            Get
                Return Occurences
            End Get
            Set(value As Double)
                Occurences = value
            End Set
        End Property

        Property GetChanceSafe As Decimal
            Get
                Return ChanceSafe
            End Get
            Set(value As Decimal)
                ChanceSafe = value
            End Set
        End Property

        Property CellContents As Char
            Get
                Return Contents
            End Get
            Set(value As Char)
                Contents = value
            End Set
        End Property

        ReadOnly Property Adjacents As List(Of Cell)
            Get
                Return AdjacentCells
            End Get
```

```
        End Property
        Property IsUseless As Boolean
            Get
                Return Useless
            End Get
            Set(value As Boolean)
                Useless = value
            End Set
        End Property

        Property IsMine As Boolean
            Get
                Return Mine
            End Get
            Set(value As Boolean)
                Mine = value
            End Set
        End Property

        ReadOnly Property GetIndex As Point
            Get
                Return Index
            End Get
        End Property
        ReadOnly Property GetLocation As Point
            Get
                Return Location
            End Get
        End Property

    End Class

End Module
```

LiveStatsForm

```
Public Class LiveStatsForm

    Private dt As New DataTable
    Sub GameEndUpdate(Outcome As String)
        GamesPlayedBox.Text += 1
        If Outcome = "Won" Then WinsBox.Text += 1
        If Outcome = "Loss" Then LossesBox.Text += 1
        Application.DoEvents()
        WinPercentBox.Text = Math.Round(WinsBox.Text / GamesPlayedBox.Text * 100,
2)
        Application.DoEvents()
    End Sub
    Sub GuessUpdate(Outcome As String)
        GuessesMadeBox.Text += 1
        If Outcome = "Success" Then SuccessfulGuessesBox.Text += 1
        If Outcome = "Fail" Then UnsuccessfulGuessesBox.Text += 1
        Application.DoEvents()
        GuessPercentBox.Text = Math.Round(SuccessfulGuessesBox.Text /
GuessesMadeBox.Text * 100, 2)
        Application.DoEvents()
    End Sub

    Sub InitTable()
        dt.Columns.Add(New DataColumn("Strategy", Type.GetType("System.String")))
        dt.Columns.Add(New DataColumn("Count", Type.GetType("System.Single")))
```

```
        For i = 0 To 3
            dt.Rows.Add(dt.NewRow)
        Next
        dt.Rows(0)("Strategy") = "Corner"
        dt.Rows(1)("Strategy") = "Guess"
        dt.Rows(2)("Strategy") = "Safe"
        dt.Rows(3)("Strategy") = "Random"
        dt.Rows(0)("Count") = 0
        dt.Rows(1)("Count") = 0
        dt.Rows(2)("Count") = 0
        dt.Rows(3)("Count") = 0
    End Sub

    Sub UpdateTable(Strategy As String)
        If dt.Columns.Count = 0 Then
            InitTable()
        End If
        If Visible Then
            Dim Row = dt.Select("Strategy = '" & Strategy & "'")
            Row(0)("Count") += 1
            UpdateLabels()
        End If
    End Sub

    Sub UpdateLabels()
        ' calculating percentages for strategies causing a loss
        If Visible Then
            CornerLossLabel.Text = "Percent Corner: " &
Math.Round(dt.Rows(0)("Count") / dt.Compute(" SUM(Count)", "") * 100, 2)
            GuessLossLabel.Text = "Percent Guess: " &
Math.Round(dt.Rows(1)("Count") / dt.Compute(" SUM(Count)", "") * 100, 2)
            SafeLossLabel.Text = "Percent Safe: " &
Math.Round(dt.Rows(2)("Count") / dt.Compute(" SUM(Count)", "") * 100, 2)
            RandomLossLabel.Text = "Percent Random: " &
Math.Round(dt.Rows(3)("Count") / dt.Compute(" SUM(Count)", "") * 100, 2)
        End If
    End Sub
```

```
    Public MoveForm As Boolean
    Public MoveForm_MousePosition As Point

    Public Sub MoveForm_MouseDown(sender As Object, e As MouseEventArgs) Handles
_
```

```
    MyBase.MouseDown ' Add more handles here (Example: PictureBox1.MouseDown)

        If e.Button = MouseButtons.Left Then
            MoveForm = True
            Me.Cursor = Cursors.NoMove2D
            MoveForm_MousePosition = e.Location
        End If

    End Sub

    Public Sub MoveForm_MouseMove(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseMove ' Add more handles here (Example: PictureBox1.MouseMove)

        If MoveForm Then
            Me.Location = Me.Location + (e.Location - MoveForm_MousePosition)
        End If

    End Sub

    Public Sub MoveForm_MouseUp(sender As Object, e As MouseEventArgs) Handles _
    MyBase.MouseUp ' Add more handles here (Example: PictureBox1.MouseUp)

        If e.Button = MouseButtons.Left Then
            MoveForm = False
            Me.Cursor = Cursors.Default
        End If

    End Sub

End Class
```

OverallStatsForm

```
Imports System.ComponentModel
Imports System.IO
Imports System.Runtime.ExceptionServices
Imports System.Text.RegularExpressions
Class OverallStatsForm
    Public HumanStats As Stats
    Public ComputerStats As Stats
    Private CurrentStats As Stats
    Private Filters As Filter()
    Private Sub OverallStatsForm_Load(sender As Object, e As EventArgs) Handles
MyBase.Load
        myStatsForm = Me
        InitiateTables()
    End Sub


    Sub InitiateTables()
        ' designs the table
        HumanStats = New Stats(Directory.GetCurrentDirectory + "\HumanStats.txt")
        ComputerStats = New Stats(Directory.GetCurrentDirectory +
"\ComputerStats.txt")
        CurrentStats = HumanStats
        StatsTable.DataSource = CurrentStats.GetTable
        StatsTable.Sort(StatsTable.Columns(2), ListSortDirection.Ascending)
        CountLabel.Text = "Count: " & CurrentStats.GetTable.DefaultView.Count
        Filters = {New NumericFilter("Time", CurrentStats, myStatsForm.MinTime,
myStatsForm.MaxTime, myStatsForm.BarTime, myStatsForm.CheckTime), New
```

```
NumericFilter("3BV", CurrentStats, myStatsForm.Min3BV, myStatsForm.Max3BV,
myStatsForm.Bar3BV, myStatsForm.Check3BV), New StringFilter("Difficulty",
myStatsForm.DifficultyBox), New StringFilter("Result", myStatsForm.ResultBox)}
        StatsTable.ColumnHeadersDefaultCellStyle.BackColor =
ColorTranslator.FromHtml("#181a1b")
        StatsTable.DefaultCellStyle.BackColor =
ColorTranslator.FromHtml("#181a1b")
        StatsTable.DefaultCellStyle.ForeColor = Color.White
        StatsTable.ColumnHeadersDefaultCellStyle.ForeColor = Color.White
        StatsTable.AutoResizeColumns()
        StatsTable.RowHeadersDefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleLeft
        StatsTable.RowHeadersDefaultCellStyle.BackColor =
ColorTranslator.FromHtml("#181a1b")
        StatsTable.RowHeadersDefaultCellStyle.ForeColor = Color.White
        StatsTable.Columns(4).Visible = False
        StatsTable.Columns(5).Visible = False
        StatsTable.ClearSelection()
    End Sub

    Sub FilterTable()
        ' collects and applies filters
        If Filters Is Nothing Then Exit Sub
        Dim TableFilter As String = ""
        For Each Filter In Filters
            If Filter.GetFilterExpression <> "" And Filter.GetFilterExpression <>
"All" Then
                If TableFilter = "" Then
                    TableFilter = Filter.GetFilterExpression
                Else
                    TableFilter += " and " & Filter.GetFilterExpression
                End If
            End If
        Next
        CurrentStats.GetTable.DefaultView.RowFilter = TableFilter
        CountLabel.Text = "Count: " & CurrentStats.GetTable.DefaultView.Count
        For Each r As DataGridViewRow In StatsTable.Rows
            r.HeaderCell.Value = CStr(r.Index + 1)
        Next

StatsTable.AutoResizeRowHeadersWidth(DataGridViewRowHeadersWidthSizeMode.AutoSize
ToAllHeaders)
        StatsTable.ClearSelection()
    End Sub

    Private Sub Reset() Handles ResetButton.Click
        For Each Filter In Filters
            Filter.Reset()
        Next
        CurrentStats.GetTable.DefaultView.RowFilter = ""
        StatsTable.ClearSelection()
    End Sub

    Private Sub ReturnToMenuButton_Click(sender As Object, e As EventArgs)
Handles ReturnToMenuButton.Click
        Hide()
        myMenuForm.Show()
    End Sub

    Private Sub SwitchViewButton_Click(sender As Object, e As EventArgs) Handles
SwitchViewButton.Click
        Reset()
        If CurrentStats Is HumanStats Then
```

```
            CurrentStats = ComputerStats
            StatsPageLabel.Text = "Computer Stats"
        ElseIf CurrentStats Is ComputerStats Then
            CurrentStats = HumanStats
            StatsPageLabel.Text = "Human Stats"
        End If
        StatsTable.DataSource = CurrentStats.GetTable
        Filters(0).StatsSwitched(CurrentStats)
        Filters(1).StatsSwitched(CurrentStats)
        CountLabel.Text = "Count: " & CurrentStats.GetTable.DefaultView.Count
        StatsTable.ClearSelection()
    End Sub

    Sub ReturnToGame(sender As Object, e As DataGridViewCellEventArgs) Handles
StatsTable.CellDoubleClick
        If e.RowIndex = -1 Then Exit Sub
        Hide()

myGameplayForm.GameFromStats(CurrentStats.GetGame(CType(StatsTable.Rows(e.RowInde
x).DataBoundItem, DataRowView).Row))
    End Sub
```

```
    Public MoveForm As Boolean
    Public MoveForm_MousePosition As Point

    Public Sub MoveForm_MouseDown(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseDown ' Add more handles here (Example: PictureBox1.MouseDown)

        If e.Button = MouseButtons.Left Then
            MoveForm = True
            Me.Cursor = Cursors.NoMove2D
            MoveForm_MousePosition = e.Location
        End If

    End Sub

    Public Sub MoveForm_MouseMove(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseMove ' Add more handles here (Example: PictureBox1.MouseMove)

        If MoveForm Then
            Me.Location = Me.Location + (e.Location - MoveForm_MousePosition)
        End If

    End Sub
```

```
    Public Sub MoveForm_MouseUp(sender As Object, e As MouseEventArgs) Handles _
    MyBase.MouseUp ' Add more handles here (Example: PictureBox1.MouseUp)

        If e.Button = MouseButtons.Left Then
            MoveForm = False
            Me.Cursor = Cursors.Default
        End If

    End Sub
End Class


Class Stats
    Private ReadOnly SourceTable As DataTable
    Private ReadOnly TextFile As String
    Private RawStats As String()
    Public ReadOnly MaxTime As Integer
    Public ReadOnly Max3BV As Integer
    Public ReadOnly MinTime As Integer
    Public ReadOnly Min3BV As Integer


    Sub New(FilePath As String)
        ' reads data from file into table
        SourceTable = New DataTable
        TextFile = FilePath
        If Not File.Exists(FilePath) Then File.Create(FilePath).Dispose()
        RawStats = File.ReadAllText(TextFile).Split(vbCrLf)
        If RawStats(0).Length = 0 Then Exit Sub
        If SourceTable.Columns.Count <> 0 Then Reset() : Exit Sub
        SourceTable.Columns.Add(New DataColumn("Difficulty",
Type.GetType("System.String")))
        SourceTable.Columns.Add(New DataColumn("3BV",
Type.GetType("System.Single")))
        SourceTable.Columns.Add(New DataColumn("Time",
Type.GetType("System.Single")))
        SourceTable.Columns.Add(New DataColumn("Result",
Type.GetType("System.String")))
        SourceTable.Columns.Add(New DataColumn("Algorithm",
Type.GetType("System.String")))
        SourceTable.Columns.Add(New DataColumn("Board",
Type.GetType("System.String")))
        For x = 0 To RawStats.Length - 1
            Dim NewRow As DataRow = SourceTable.NewRow
            Dim CurrentLine = RawStats(x).Split(",")
            If CurrentLine.Length = 1 Then Continue For
            NewRow("Difficulty") = DisplayDifficulty(CurrentLine(0))
            NewRow("3BV") = Single.Parse(CurrentLine(1))
            NewRow("Time") = Single.Parse(CurrentLine(2))
            NewRow("Result") = CurrentLine(3)
            NewRow("Algorithm") = CurrentLine(4)
            NewRow("Board") = Join(CurrentLine.Skip(5).ToArray, ",")
            SourceTable.Rows.Add(NewRow)
        Next
        MaxTime = SourceTable.Compute("Max([Time])", "")
        MinTime = SourceTable.Compute("Min([Time])", "")
        Max3BV = SourceTable.Compute("Max([3BV])", "")
        Min3BV = SourceTable.Compute("Min([3BV])", "")
    End Sub


    Function DisplayDifficulty(Line As String) As String
```

```
        If Line = "Beginner" Then
            Return "Beginner"
        ElseIf Line = "Intermediate" Then
            Return "Intermediate"
        ElseIf Line = "Expert" Then
            Return "Expert"
        Else
            Return "" & Line
        End If
        Return Nothing
    End Function

    Function GetGame(RowToFind As DataRow) As String
        Dim Reader As StreamReader =
My.Computer.FileSystem.OpenTextFileReader(TextFile)
        Dim RowIndex As Integer = SourceTable.Rows.IndexOf(RowToFind)
        Dim CurrentIndex As Integer = 0
        Do Until CurrentIndex = RowIndex
            CurrentIndex += 1
            Reader.ReadLine()
        Loop
        Dim Game As String = Reader.ReadLine
        Return Game.Substring(Regex.Matches(Game, ",")(4).Index + 1)
    End Function
    ReadOnly Property GetTable As DataTable
        Get
            Return SourceTable
        End Get
    End Property

End Class

Class Filter
    Protected FilterName As String
    Protected FilterExpression As String
    Sub New(Name As String)
        FilterExpression = ""
        FilterName = Name
    End Sub
    Overridable Sub Reset()
    End Sub

    Overridable Sub StatsSwitched(CurrentStats As Stats)
    End Sub

    Protected Sub FilterChanged()
        myStatsForm.FilterTable()
    End Sub

    Protected Overridable Sub UpdateFilter()
    End Sub
    ReadOnly Property GetFilterExpression As String
        Get
            Return FilterExpression
        End Get
    End Property
End Class

Class NumericFilter
    Inherits Filter
    Private WithEvents LessThanChecked As CheckBox
    Private MinimumLabel As Label
    Private MaximumLabel As Label
```

```
    Private WithEvents FilterBar As TrackBar

    Sub New(Name As String, CurrentStats As Stats, MinLabel As Label, MaxLabel As
Label, Bar As TrackBar, LessThanCheckBox As CheckBox)
        MyBase.New(Name)
        MinimumLabel = MinLabel
        MaximumLabel = MaxLabel
        FilterBar = Bar
        LessThanChecked = LessThanCheckBox
        If Name = "Time" Then FilterBar.Maximum = CurrentStats.MaxTime Else
FilterBar.Maximum = CurrentStats.Max3BV
        If Name = "Time" Then FilterBar.Minimum = CurrentStats.MinTime Else
FilterBar.Minimum = CurrentStats.Min3BV
        MinimumLabel.Text = FilterBar.Minimum
        MaximumLabel.Text = FilterBar.Maximum
        LessThanChecked.Checked = False
    End Sub

    Protected Overrides Sub UpdateFilter() Handles FilterBar.MouseUp,
LessThanChecked.CheckedChanged
        If LessThanChecked.Checked Then
            FilterExpression = "[" & FilterName & "] <= " & FilterBar.Value
        Else
            FilterExpression = "[" & FilterName & "] >= " & FilterBar.Value
        End If
        FilterChanged()
    End Sub

    Private Sub UpdateBar() Handles FilterBar.ValueChanged,
LessThanChecked.CheckedChanged
        If LessThanChecked.Checked Then
            MaximumLabel.Text = FilterBar.Value
            MinimumLabel.Text = FilterBar.Minimum
        Else
            MinimumLabel.Text = FilterBar.Value
            MaximumLabel.Text = FilterBar.Maximum
        End If
    End Sub

    Overrides Sub StatsSwitched(CurrentStats As Stats)
        If FilterName = "Time" Then FilterBar.Maximum = CurrentStats.MaxTime Else
FilterBar.Maximum = CurrentStats.Max3BV
        If FilterName = "Time" Then FilterBar.Minimum = CurrentStats.MinTime Else
FilterBar.Minimum = CurrentStats.Min3BV
        MinimumLabel.Text = FilterBar.Minimum
        MaximumLabel.Text = FilterBar.Maximum
        LessThanChecked.Checked = False
        Reset()
    End Sub

    Overrides Sub Reset()
        FilterExpression = ""
        FilterBar.Value = FilterBar.Minimum
        MinimumLabel.Text = FilterBar.Minimum
        MaximumLabel.Text = FilterBar.Maximum
        LessThanChecked.Checked = False
    End Sub
End Class

Class StringFilter
    Inherits Filter
    Private WithEvents OptionsBox As ComboBox
```

```
    Sub New(Name As String, Options As ComboBox)
        MyBase.New(Name)
        OptionsBox = Options
        OptionsBox.SelectedItem = "All"
    End Sub

    Protected Overrides Sub UpdateFilter() Handles
OptionsBox.SelectedIndexChanged
        If OptionsBox.SelectedItem IsNot "All" Then
            FilterExpression = FilterName & " LIKE '*" & OptionsBox.SelectedItem
& "*'"
        Else
            FilterExpression = ""
        End If
        FilterChanged()
    End Sub

    Overrides Sub Reset()
        OptionsBox.SelectedIndex = 0
        OptionsBox.SelectedItem = "All"
    End Sub
End Class
```

CustomBoardForm

```
Imports System.IO
Public Class CustomBoardForm
    Private TitleBarSize As Integer = Height - ClientRectangle.Height
    Private BoardBitmap As Bitmap
    Private Img As Boolean
    Private CustomBoardArray(,) As Rectangle
    Private CustomStateArray(,) As Char
    Private MouseLocation As Point

    Public Sub Form_Unload(sender As Object, e As EventArgs) Handles Me.Closing
        myMenuForm.Show()
    End Sub
    Private Sub GenerateBoard() Handles GenerateBoardButton.Click
        If Not Integer.TryParse(XSizeBox.Text, vbNull) OrElse Not
Integer.TryParse(YSizeBox.Text, vbNull) Then ErrorMessage("Invalid value
entered") : Exit Sub
        If XSizeBox.Text > 50 OrElse YSizeBox.Text > 50 Then ErrorMessage("Value
cannot be over 50") : Exit Sub
        If XSizeBox.Text <= 0 OrElse YSizeBox.Text <= 0 Then
ErrorMessage("Invalid Size") : Exit Sub

        BoardBitmap = New Bitmap(CInt(XSizeBox.Text) * 16, CInt(YSizeBox.Text) *
16)
        SolvedBoardBox.Width = XSizeBox.Text * 16
        SolvedBoardBox.Height = YSizeBox.Text * 16
        ReDim CustomBoardArray(XSizeBox.Text - 1, YSizeBox.Text - 1)
        ReDim CustomStateArray(XSizeBox.Text - 1, YSizeBox.Text - 1)
        Using g = Graphics.FromImage(BoardBitmap)
            For x = 0 To XSizeBox.Text - 1
                For y = 0 To YSizeBox.Text - 1
                    CustomBoardArray(x, y) = New Rectangle(x * 16, y * 16, 15,
15)
                    CustomStateArray(x, y) = "u"
                    g.DrawImage(My.Resources.UnknownCell, CustomBoardArray(x,
y).Location)
                Next
```

```
            Next
        End Using
        SolvedBoardBox.Image = BoardBitmap
        ResizeForm()
        SolvedBoardBox.Show()
        Img = False
    End Sub

    Sub ImageSizings(Board As Bitmap)
        XSizeBox.Text = Board.Width / 16
        YSizeBox.Text = Board.Height / 16
        ReDim CustomBoardArray(XSizeBox.Text - 1, YSizeBox.Text - 1)
        ReDim CustomStateArray(XSizeBox.Text - 1, YSizeBox.Text - 1)
    End Sub


    Sub DisplaySolvedBoard(Board As Bitmap)
        SafeLabel.Text = "Safe"
        GuessLabel.Text = "Guess"
        MineLabel.Text = "Mine"
        Dim Solved As SolvedBoard = SolveBoard(Board, Img)
        If Solved Is Nothing Then
            If Img Then ErrorMessage("Couldn't find board or Board is Invalid")
Else ErrorMessage("Invalid Board")
            Exit Sub
        End If
        SolvedBoardBox.Size = New Size(Solved.Board.Width, Solved.Board.Height)
        BoardBitmap = Solved.Board
        Using g = Graphics.FromImage(BoardBitmap)
            If Solved.SafeCells.Count <> 0 Then
                SafeLabel.Text = "Safe (Count: " & Solved.SafeCells.Count & ")"
                For Each SafeCell In Solved.SafeCells
                    g.DrawImage(My.Resources.SafeCell, SafeCell.GetIndex.X * 16,
SafeCell.GetIndex.Y * 16)
                    If Not Img Then CustomStateArray(SafeCell.GetIndex.X,
SafeCell.GetIndex.Y) = "S"
                Next
            ElseIf Solved.Guess.GetCell IsNot Nothing Then
                GuessLabel.Text = "Guess (" &
Math.Round(Solved.Guess.GetChanceSafe, 2) & "& Chance)"
                g.DrawImage(My.Resources.GuessCell,
Solved.Guess.GetCell.GetIndex.X * 16, Solved.Guess.GetCell.GetIndex.Y * 16)
                If Not Img Then CustomStateArray(Solved.Guess.GetCell.GetIndex.X,
Solved.Guess.GetCell.GetIndex.Y) = "G"
            End If
            For Each MineCell In Solved.MinesList
                MineLabel.Text = "Mine (Count: " & Solved.MinesList.Count & ")"
                g.DrawImage(My.Resources.MineCell, MineCell.GetIndex.X * 16,
MineCell.GetIndex.Y * 16)
                If Not Img AndAlso CustomStateArray(MineCell.GetIndex.X,
MineCell.GetIndex.Y) <> "f" Then CustomStateArray(MineCell.GetIndex.X,
MineCell.GetIndex.Y) = "M"
            Next
        End Using
        SolvedBoardBox.Image = BoardBitmap
        ResizeForm()
        SolvedBoardBox.Show()
    End Sub
    Private Sub ImageButton_Click(sender As Object, e As EventArgs) Handles
ImageButton.Click
        Img = True
        If Clipboard.GetImage Is Nothing Then ErrorMessage("No image found") :
Img = False : Exit Sub
```

```
        DisplaySolvedBoard(Clipboard.GetImage())
    End Sub

    Private Sub ResizeForm()
        'makes sure the window size is proportional to the board size
        UserOptionsBox.Location = New Point(SolvedBoardBox.Location.X +
SolvedBoardBox.Width + 10, UserOptionsBox.Location.Y)
        Size = New Size(UserOptionsBox.Location.X + UserOptionsBox.Width + 10,
Math.Max(UserOptionsBox.Height + 43, SolvedBoardBox.Height + 43) + 43)
        SolvedBoardBox.Location = New Point(0, (Size.Height -
SolvedBoardBox.Height) / 2)
        UserOptionsBox.Location = New Point(UserOptionsBox.Location.X,
(Size.Height - UserOptionsBox.Height) / 2)
    End Sub
    Private Sub SolvedBoardBox_Click(sender As Object, e As MouseEventArgs)
Handles SolvedBoardBox.Click
        If Not Img Then
            For Each Rect In CustomBoardArray
                If Rect.Contains(e.Location) Then
                    If e.Button = MouseButtons.Left Then
                        Using ChoiceForm As New CellChoiceForm
                            ChoiceForm.ShowDialog()
                            If ChoiceForm.ChosenCell IsNot Nothing Then
DrawCell(ChoiceForm.ChosenCell, ChoiceForm.ChosenState, Rect)
                        End Using
                    Else
                        If CustomStateArray(Rect.X / 16, Rect.Y / 16) <> "f" Then
                            DrawCell(My.Resources.FlagCell, "f", Rect)
                        Else
                            DrawCell(My.Resources.UnknownCell, "u", Rect)
                        End If
                    End If
                End If
            Next
            SolvedBoardBox.Image = BoardBitmap
        End If
    End Sub
    Sub DrawCell(ChosenCell As Bitmap, ChosenState As Char, Rect As Rectangle)
        Using g = Graphics.FromImage(BoardBitmap)
            g.DrawImage(ChosenCell, Rect.Location)
            CustomStateArray(Rect.X / 16, Rect.Y / 16) = ChosenState
        End Using
        SolvedBoardBox.Image = BoardBitmap
    End Sub
    Private Sub ResetButton_Click(sender As Object, e As EventArgs) Handles
ResetButton.Click
        If SolvedBoardBox.Image IsNot Nothing Then
            If Img Then
                GenerateBoard()
                SafeLabel.Text = "Safe"
                MineLabel.Text = "Mine"
                GuessLabel.Text = "Guess"
            Else
                XSizeBox.Text = CustomBoardArray.GetUpperBound(0) + 1
                YSizeBox.Text = CustomBoardArray.GetUpperBound(1) + 1
                GenerateBoard()
                SafeLabel.Text = "Safe"
                MineLabel.Text = "Mine"
                GuessLabel.Text = "Guess"
            End If
        End If
    End Sub
```

```
    Private Sub ReturnToMenuButton_Click(sender As Object, e As EventArgs)
Handles ReturnToMenuButton.Click
        Hide()
        myMenuForm.Show()
    End Sub

    Private Sub SolveButton_Click(sender As Object, e As EventArgs) Handles
SolveButton.Click
        If Not Img Then
            If SolvedBoardBox.Image IsNot Nothing Then
                ClearShading()
                DisplaySolvedBoard(SolvedBoardBox.Image)
            Else
                ErrorMessage("No Board")
            End If
        Else
            ErrorMessage("Please Generate a Custom Board")
        End If
    End Sub

    Private Sub ClearShading() Handles ClearShadingButton.Click
        If SolvedBoardBox.Image IsNot Nothing Then
            Dim ClearedBoard As Bitmap = SolvedBoardBox.Image
            Using g = Graphics.FromImage(ClearedBoard)
                For x = 0 To CustomStateArray.GetUpperBound(0)
                    For y = 0 To CustomStateArray.GetUpperBound(1)
                        If CustomStateArray(x, y) = "S" OrElse
CustomStateArray(x, y) = "G" OrElse CustomStateArray(x, y) = "M" Then
                            g.DrawImage(My.Resources.UnknownCell, New Point(x *
16, y * 16))
                        ElseIf CustomStateArray(x, y) = "f" Then
                            g.DrawImage(My.Resources.FlagCell, New Point(x * 16,
y * 16))
                        End If
                    Next
                Next
            End Using
            SolvedBoardBox.Image = ClearedBoard
        End If
    End Sub

    Private Sub SolvedBoardBox_Hover(sender As Object, e As MouseEventArgs)
Handles SolvedBoardBox.MouseMove
        If Not Img Then SolvedBoardBox.Cursor = Cursors.Hand Else
SolvedBoardBox.Cursor = Cursors.Default
        SolvedBoardBox.Focus()
        MouseLocation = e.Location
    End Sub

    Private Sub SolvedBoardBox_KeyDown(sender As Object, e As KeyEventArgs)
Handles SolvedBoardBox.KeyDown
        For Each Rect In CustomBoardArray
            If Rect.Contains(MouseLocation) Then
                Select Case e.KeyCode
                    Case Keys.D0, Keys.Space
                        DrawCell(My.Resources.BlankCell, "0", Rect)
                    Case Keys.D1
                        DrawCell(My.Resources._1Cell, "1", Rect)
                    Case Keys.D2
                        DrawCell(My.Resources._2Cell, "2", Rect)
                    Case Keys.D3
                        DrawCell(My.Resources._3Cell, "3", Rect)
                    Case Keys.D4
```

```
                    DrawCell(My.Resources._4Cell, "4", Rect)
                Case Keys.D5
                    DrawCell(My.Resources._5Cell, "5", Rect)
                Case Keys.D6
                    DrawCell(My.Resources._6Cell, "6", Rect)
                Case Keys.D7
                    DrawCell(My.Resources._7Cell, "7", Rect)
                Case Keys.D8
                    DrawCell(My.Resources._8Cell, "8", Rect)
                Case Keys.U
                    DrawCell(My.Resources.UnknownCell, "u", Rect)
                Case Keys.M
                    DrawCell(My.Resources.FlagCell, "f", Rect)
                Case Keys.F
                    DrawCell(My.Resources.FlagCell, "f", Rect)
            End Select
        End If
    Next



End Sub

Private Sub SaveImageButton_Click(sender As Object, e As EventArgs) Handles
SaveImageButton.Click
    If BoardBitmap IsNot Nothing Then
        SaveImage.Filter = "Image Files(*.png)|*.png"
        If SaveImage.ShowDialog() = Windows.Forms.DialogResult.OK Then
            BoardBitmap.Save(SaveImage.FileName)
        End If
    End If
End Sub
```

```
Public MoveForm As Boolean
Public MoveForm_MousePosition As Point

Public Sub MoveForm_MouseDown(sender As Object, e As MouseEventArgs) Handles
_
MyBase.MouseDown ' Add more handles here (Example: PictureBox1.MouseDown)

    If e.Button = MouseButtons.Left Then
        MoveForm = True
        Me.Cursor = Cursors.NoMove2D
        MoveForm_MousePosition = e.Location
    End If
```

```
    End Sub

    Public Sub MoveForm_MouseMove(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseMove ' Add more handles here (Example: PictureBox1.MouseMove)

        If MoveForm Then
            Me.Location = Me.Location + (e.Location - MoveForm_MousePosition)
        End If

    End Sub

    Public Sub MoveForm_MouseUp(sender As Object, e As MouseEventArgs) Handles _
    MyBase.MouseUp ' Add more handles here (Example: PictureBox1.MouseUp)

        If e.Button = MouseButtons.Left Then
            MoveForm = False
            Me.Cursor = Cursors.Default
        End If

    End Sub

End Class
```

CustomGameplayModule

```
Module CustomGamePlayModule

    Function SolveBoard(UserImage As Bitmap, Img As Boolean) As SolvedBoard
        Dim Board As New CustomBoard(UserImage, Img)
        If Board.GetBoard Is Nothing Then Return Nothing ' no board would mean
invalid
        Dim myCustomGame As New CustomGame(Board)
        myCustomGame.SolveBoard()
        Return myCustomGame.GetSolved
    End Function

    Class CustomGame
        Inherits Game
        Private Board As CustomBoard
        Private Solved As SolvedBoard
        Sub New(CurrentBoard As CustomBoard)
            Board = CurrentBoard
        End Sub

        Sub SolveBoard()
            Cells = GetCells()
            If Cells Is Nothing Then ErrorMessage("Game Already Finished") : Exit
Sub
            Guess = New BestGuessCell
            For Each Cell In Cells
                Cell.GenerateAdjacents(Cells, Board.GetBoard.Width / 16 - 1,
Board.GetBoard.Height / 16 - 1)
                If Cell.Adjacents.Count < Val(Cell.CellContents) Then Solved =
Nothing : Exit Sub
                Cell.CheckIfMine()
                Cell.CheckIfUseless()
            Next
            Dim SafeCount As Integer = -1
```

```
            Dim myConstraintSolver As ConstraintSolver
            Dim Looped As Boolean = False
            Do
                If SafeCount <> -1 Then Looped = True ' loops once to ensure all
safe cells are located
                SafeCount = SafeCells.Count
                For Each Cell In Cells
                    If Cell.SimpleFinder IsNot Nothing Then
                        For Each SafeCell In Cell.SimpleFinder()
                            If Not SafeCells.Contains(SafeCell) Then
SafeCells.Add(SafeCell)
                        Next
                    Else
                        Solved = Nothing
                        Exit Sub
                    End If

                Next
                myConstraintSolver = New ConstraintSolver(Cells,
DeepClone(SafeCells), True)
                myConstraintSolver.FindNonTrivialSolutions()
                If myConstraintSolver.GetSafeCells Is Nothing Then Solved =
Nothing : Exit Sub
                For Each SafeCell In myConstraintSolver.GetSafeCells
                    If SafeCells.Find(Function(x) x.GetIndex = SafeCell.GetIndex)
Is Nothing Then SafeCells.Add(SafeCell)
                Next
                Dim ReplacementSafeCells As New List(Of Cell)
                For Each Cell In SafeCells
                    ReplacementSafeCells.Add(Cells(Cell.GetIndex.X,
Cell.GetIndex.Y))
                Next
                SafeCells = ReplacementSafeCells
                For Each Cell In Cells
                    If Cell.Adjacents.Except(SafeCells).Count =
Val(Cell.CellContents) Then
                        For Each Mine In Cell.Adjacents.Except(SafeCells)
                            If Mine.IsMine = False Then Mine.IsMine = True
                        Next
                    End If
                Next
            Loop Until SafeCount = SafeCells.Count And Looped
            If SafeCells.Count = 0 AndAlso myConstraintSolver.GetGuess IsNot
Nothing Then Guess = myConstraintSolver.GetGuess
            Solved = New SolvedBoard(SafeCells, Board.GetBoard, Guess)
            For Each Cell In Cells
                If Cell.IsMine Then
                    Solved.MinesList.Add(Cell)
                End If
            Next
            For Each Cell In Cells
                If Cell.Adjacents.FindAll(Function(x) Not
SafeCells.Contains(x)).Count < Val(Cell.CellContents) Then Solved = Nothing :
Exit Sub ' checking that the board is valid
            Next
        End Sub

        Overrides Function GetCells() As Cell(,)
            Dim Cells(Board.GetBoard.Width / 16 - 1, Board.GetBoard.Height / 16 -
1) As Cell
            Dim xCounter
            Dim yCounter = 0
            For y = 8 To Board.GetBoard.Height - 8 Step 16
```

```
                xCounter = 0
                For x = 8 To Board.GetBoard.Width - 8 Step 16
                    Cells(xCounter, yCounter) = New Cell(Board.GetBoard, New
Point(x, y), New Point(xCounter, yCounter))
                    If Cells(xCounter, yCounter).CellContents = Nothing Then
Return Nothing
                    If xCounter <> Board.GetBoard.Width / 16 - 1 Then
                        xCounter += 1
                    Else
                        xCounter = 0
                    End If
                Next
                yCounter += 1
            Next
            Return Cells
        End Function

        ReadOnly Property GetSolved As SolvedBoard
            Get
                Return Solved
            End Get
        End Property
    End Class
    Class CustomBoard
        Private Board As Bitmap
        Private CellBitmaps() As Bitmap = {My.Resources.BlankCell,
My.Resources.UnknownCell, My.Resources._1Cell, My.Resources._2Cell,
                                           My.Resources._3Cell,
My.Resources._4Cell, My.Resources._5Cell,
                                           My.Resources._6Cell,
My.Resources._7Cell, My.Resources._8Cell}
        Private StartCoord As Point
        Private EndCoord As Point
        Sub New(UserImage As Bitmap, Img As Boolean)
            If Img Then
                StartCoord = FindStart(UserImage)
                If StartCoord.X > UserImage.Width Then Exit Sub
                EndCoord = FindEnd(UserImage)
                If EndCoord.X > UserImage.Width Then Exit Sub
            Else
                StartCoord = New Point(0, 0)
                EndCoord = New Point(UserImage.Width - 1, UserImage.Height - 1)
            End If
            Dim BoardRect As New Rectangle(StartCoord.X, StartCoord.Y, EndCoord.X
- StartCoord.X + 1, EndCoord.Y - StartCoord.Y + 1)
            Board = New Bitmap(BoardRect.Width, BoardRect.Height)
            Using grp = Graphics.FromImage(Board)
                grp.DrawImage(UserImage, New Rectangle(0, 0, BoardRect.Width,
BoardRect.Height), BoardRect, GraphicsUnit.Pixel)
            End Using
            If Img Then myCustomGameForm.ImageSizings(Board)
        End Sub
        Private Function FindStart(UserImage As Bitmap) As Point
            ' start of board found by comparing colours
            For y As Integer = 0 To UserImage.Height -
My.Resources.UnknownCell.Height - 1

                For x As Integer = 0 To UserImage.Width -
My.Resources.UnknownCell.Width - 1
                    Dim clr As Color = UserImage.GetPixel(x, y)
                    For Each Cell In CellBitmaps
```

```
                        If clr = Cell.GetPixel(0, 0) AndAlso
IsInnerImage(UserImage, Cell, x, y) Or clr = Cell.GetPixel(0, 0) AndAlso
IsInnerImage(UserImage, Cell, x, y) Then
                            Return New Point(x, y)
                        End If
                    Next
                Next
            Next
            Return New Point(UserImage.Width + 1, UserImage.Height + 1) ' this
would indicate no board is found
        End Function
        Private Function FindEnd(UserImage As Bitmap) As Point
            ' same process a start but in reverse
            For y = UserImage.Height - 1 To My.Resources.UnknownCell.Height Step
-1
                For x = UserImage.Width - 1 To My.Resources.UnknownCell.Width
Step -1
                    Dim clr As Color = UserImage.GetPixel(x, y)
                    For Each Cell In CellBitmaps
                        If clr = Cell.GetPixel(Cell.Width - 1, Cell.Height - 1)
AndAlso IsInnerImage(UserImage, Cell, x - Cell.Width + 1, y - Cell.Height + 1) Or
                            clr = Cell.GetPixel(Cell.Width - 1, Cell.Height - 1)
AndAlso IsInnerImage(UserImage, Cell, x - Cell.Width + 1, y - Cell.Height + 1)
Then
                            Return New Point(x, y)
                        End If
                    Next
                Next

            Next
            Return New Point(UserImage.Width + 1, UserImage.Height + 1)
        End Function
        Private Function IsInnerImage(UserImage As Bitmap, CellBitmap As Bitmap,
left As Integer, top As Integer) As Boolean
            For y As Integer = top To top + CellBitmap.Height - 1

                For x As Integer = left To left + CellBitmap.Width - 1
                    If UserImage.GetPixel(x, y) <> CellBitmap.GetPixel(x - left,
y - top) Then Return False
                Next
            Next

            Return True
        End Function
        ReadOnly Property GetBoard As Bitmap
            Get
                Return Board
            End Get
        End Property
    End Class

    Public Class SolvedBoard
        Public MinesList As New List(Of Cell)
        Public SafeCells As New List(Of Cell)
        Public Guess As New BestGuessCell
        Public Board As Bitmap
        Sub New(CellList As List(Of Cell), GameBoard As Bitmap, BestGuess As
BestGuessCell)
            Board = GameBoard
            If CellList.Count <> 0 Then SafeCells = CellList Else Guess =
BestGuess
        End Sub
    End Class
```

```
End Module
```

CellChoiceForm

```
Public Class CellChoiceForm
    Public ChosenCell As Image
    Public ChosenState As Char
    Public Sub CellChosen(sender As Object, e As EventArgs) Handles
BlankCell.Click, Cell1.Click, Cell2.Click, Cell3.Click,
                                                                Cell4.Click,
Cell5.Click, Cell6.Click, Cell7.Click, Cell8.Click, UnknownCell.Click
        If sender Is BlankCell Then
            ChosenState = "0"
            ChosenCell = My.Resources.BlankCell
        ElseIf sender Is Cell1 Then
            ChosenState = "1"
            ChosenCell = My.Resources._1Cell
        ElseIf sender Is Cell2 Then
            ChosenState = "2"
            ChosenCell = My.Resources._2Cell
        ElseIf sender Is Cell3 Then
            ChosenState = "3"
            ChosenCell = My.Resources._3Cell
        ElseIf sender Is Cell4 Then
            ChosenState = "4"
            ChosenCell = My.Resources._4Cell
        ElseIf sender Is Cell5 Then
            ChosenState = "5"
            ChosenCell = My.Resources._5Cell
        ElseIf sender Is Cell6 Then
            ChosenState = "6"
            ChosenCell = My.Resources._6Cell
        ElseIf sender Is Cell7 Then
            ChosenState = "7"
            ChosenCell = My.Resources._7Cell
        ElseIf sender Is Cell8 Then
            ChosenState = "8"
            ChosenCell = My.Resources._8Cell
        ElseIf sender Is UnknownCell Then
            ChosenState = "u"
            ChosenCell = My.Resources.UnknownCell
        End If

        DialogResult = DialogResult.OK

    End Sub




    Public MoveForm As Boolean
    Public MoveForm_MousePosition As Point

    Public Sub MoveForm_MouseDown(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseDown ' Add more handles here (Example: PictureBox1.MouseDown)

        If e.Button = MouseButtons.Left Then
            MoveForm = True
```

```
            Me.Cursor = Cursors.NoMove2D
            MoveForm_MousePosition = e.Location
        End If

    End Sub

    Public Sub MoveForm_MouseMove(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseMove ' Add more handles here (Example: PictureBox1.MouseMove)

        If MoveForm Then
            Me.Location = Me.Location + (e.Location - MoveForm_MousePosition)
        End If

    End Sub

    Public Sub MoveForm_MouseUp(sender As Object, e As MouseEventArgs) Handles _
    MyBase.MouseUp ' Add more handles here (Example: PictureBox1.MouseUp)

        If e.Button = MouseButtons.Left Then
            MoveForm = False
            Me.Cursor = Cursors.Default
        End If

    End Sub

End Class
```

TrainerForm

```
Public Class TrainerForm
    Private Sub TrainerForm_Load(sender As Object, e As EventArgs) Handles
MyBase.Load
        myStatsForm.InitiateTables()
    End Sub



    Sub StartGame(sender As Button, e As MouseEventArgs) Handles
SimpleButton.Click, NonTrivialButton.Click, BruteForceButton.Click,
GuessButton.Click
        Dim r As New Random
        Dim RandomGame As DataRow
        Dim Difficulty As String = ""

        Select Case sender.Name
            Case "SimpleButton"
                Difficulty = "Simple"
            Case "NonTrivialButton"
                Difficulty = "Non-Trivial"
            Case "BruteForceButton"
                Difficulty = "Brute Force"
            Case "GuessButton"
                Difficulty = "Guess"
        End Select
        Dim Count As Integer = 1000
        Do
            RandomGame = myStatsForm.ComputerStats.GetTable.Rows(r.Next(0,
myStatsForm.ComputerStats.GetTable.Rows.Count - 1))
            Count -= 1
        Loop Until RandomGame.ItemArray(4) = Difficulty Or Count = 0
```

```
            If Count <> 0 Then
                myGameplayForm.Hide()

myGameplayForm.GameFromStats(myStatsForm.ComputerStats.GetGame(RandomGame))
                myGameplayForm.Show()
            End If
        End Sub
```

```
    Public MoveForm As Boolean
    Public MoveForm_MousePosition As Point

    Public Sub MoveForm_MouseDown(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseDown ' Add more handles here (Example: PictureBox1.MouseDown)

        If e.Button = MouseButtons.Left Then
            MoveForm = True
            Me.Cursor = Cursors.NoMove2D
            MoveForm_MousePosition = e.Location
        End If

    End Sub

    Public Sub MoveForm_MouseMove(sender As Object, e As MouseEventArgs) Handles
_
    MyBase.MouseMove ' Add more handles here (Example: PictureBox1.MouseMove)

        If MoveForm Then
            Me.Location = Me.Location + (e.Location - MoveForm_MousePosition)
        End If

    End Sub

    Public Sub MoveForm_MouseUp(sender As Object, e As MouseEventArgs) Handles _
    MyBase.MouseUp ' Add more handles here (Example: PictureBox1.MouseUp)

        If e.Button = MouseButtons.Left Then
            MoveForm = False
            Me.Cursor = Cursors.Default
        End If

    End Sub
End Class
```

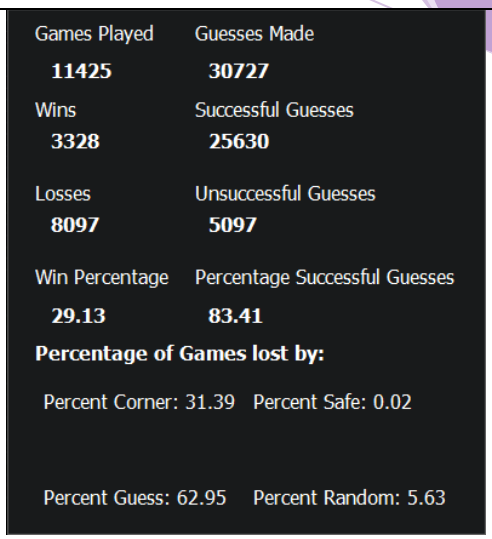
Testing

All tests are evidenced in the following video, and timestamps are provided for each test:

https://youtu.be/FgOHCoBvvaA

Test Number	Description	Expected Outcome	Evidence and Actual Outcome
1	Generate a Beginner, Intermediate and Expert board using their respective buttons.	All three boards generated successfully.	0:00 – 0:09 As Intended
2	Generate a 20x20/100 board.	Board successfully generated.	0:09 – 0:18 As Intended
3	Generate a 0x0/0 board.	Nothing generated as board is invalid.	0:18 – 0:26 As Intended
4	Generate a 100x100/1000 board.	Values of width and height should automatically max out at 50, so this board cannot be generated.	0:26 – 0:36 As Intended
5	Generate a 20x20/400 board.	Nothing generated as board has too many mines.	00:36 – 0:46 As Intended
6	Click first cell on board, then reset the board. Repeat this 10 times.	First cell should always be safe, when board is reset, it should be randomized (new mine locations) but the same dimensions.	0:46 – 1:07 As Intended
7	Open a blank cell.	All adjacent cells should be opened. If any were blank then this should happen recursively.	1:07 – 1:10 As Intended
8	Chord	After clicking on	1:10 – 1:17

	successfully by clicking on a numbered cell which has the correct number of flags adjacent to it.	the cell, each non-flagged adjacent cell should be opened around it, any blank cells should be handled correctly.	As Intended
9	Chord unsuccessfully by flagging cells which are not mines, then clicking a numbered cell and opening a mine as a result.	After clicking the cell, the game should end as a mine will have been opened.	1:17 – 1:24 As Intended
10	Win a game	After opening the final non-mine cell, the user should be informed they have won by automatically flagging all mines and changing the reset button image.	1:24 – 1:37 As Intended
11	Lose a game	After opening a mine cell, the user should be informed they have lost by automatically opening all mines and changing the reset button image.	1:37 – 1:44 As Intended
12	Show a minute of the solver running.	The solver should play through a board, with no significant breaks (longer than 5 seconds) for computation and when a game finishes, it should be automatically reset.	1:44 – 2:43 As Intended (Note that the Live Stats had been running previously, hence why it said 12 games had already been played)

13	Record the results of at least 10,000 expert games	The solver should have a win rate of above 25% on expert difficulty.	 As Intended
14	Play a game, then show it in the stats file.	After the game is finished, the stats file should have had a line appended to it corresponding to that game.	2:43 – 2:56 As Intended
15	Show live tracker for 15 seconds.	The live tracker should refresh its display after every game with the correct updated values for each piece of data.	2:56 – 3:12 As Intended
16	Generate a custom board of size 20x20.	The board should be generated.	3:12 – 3:18 As Intended
17	Generate a custom board of size 0x0.	An error message should be displayed as this board size is invalid.	3:18 – 3:24 As Intended
18	Generate a custom board of size 100x100	An error message should be displayed as this board size is too large.	3:24 – 3:32 As Intended
19	Solve a valid custom board.	The board should be highlighted with the location of any mines and	3:32 – 3:41 As Intended

		any safe cells/guesses.	
20	Attempt to solve an invalid board.	An error message should be displayed telling the user they inputted an invalid board.	3:41 – 3:49 As Intended
21	Save an image of a board to the computer.	A file explorer tab should be opened, allowing the user to successfully name and save an image of the board to the computer.	3:49 – 4:03 As Intended
22	Use an image to recreate a valid board and solve it from clipboard.	The board contained in the image from clipboard should be displayed on screen and solved, with all mines and safe cells/guesses highlighted.	4:03 – 4:10 As intended
23	Show a game being recreated from stats.	When a game in the stats table is double clicked, a exact recreation of that board should be created, which can be played. If the same game is clicked again, the exact same board should be created.	4:10 – 4:26 As intended
24	Show all 4 trainer buttons generating a new board.	After each button is clicked, a new randomized board should be generated of the correct difficulty.	4:26 – 4:34

Evaluation

Evaluating my Objectives

Required Objectives

1. **The program should be able to generate any size of board, up to and including 50x50.**

   My program can successfully generate any board up to 50x50 and rejects any board bigger or smaller as shown in **Tests 1 - 4**. If I were to improve upon this objective, I could find a way to allow for larger board sizes, although this wouldn't have a significant effect as 50x50 is large enough for almost all purposes.

2. **The program should allow any number of mines on this board, provided it is less than the number of cells on the board.**

   My program will only generate a board if the mine count is less than the total number of cells on the board as shown in **Test 5**. This can't really be improved upon, but could be added to the custom games also, if I added a mine count for those boards (more detail later).

3. **When either a human or computer clicks the board for the first time in a game, it should always be safe.**

   Having tested this thoroughly (no occurrences over the games played in **Test 13**), I am confident this objective has been met, with evidence shown in **Test 6**. If I were to improve upon this objective, I could add an option to toggle this on or off. Almost all implementations of Minesweeper have a safe first click, but there are some that don't, so an option would allow this program to imitate both.

4. **When blank cell is opened, all adjacent cells should also be opened, this should be recursive for any blank cells opened.**

   This has also been tested thoroughly, with evidence in **Test 7**, and functions as intended in the program. There isn't a clear way to improve upon this objective.

5. **When a numbered cell is clicked, if there are the correct number of flags adjacent to it, then all non-mine cells adjacent should be opened (Chording).**

   Chording is possible in my implementation, and functions as intended (shown in **Test 8 and 9**). One improvement that could be made to improve chording is adding an animation. In many versions of the game, if you attempted to chord, but there are not enough mines surrounding the cell you click, then then animation will play to show you the remaining cells which could be flagged to allow for chording on that cell. This feature is almost entirely for aesthetical purposes but could be added to improve user experience.

6. **When all non-mine cells have been opened, the user should be informed that the game has finished, and they have won.**

The user is informed that the game has been won in multiple ways, all mines on the board become flagged, and the icon of the reset button changes to have sunglasses (as it did in the original version of the game). This functionality is shown in **Test 10**. The only improvement that could be a made is a more obvious indicator that the user has won, although this would only help for the first few games, then it would likely get in the way.

7. **When a mine cell is opened, the user should be informed that the game has finished, and they have lost.**

   The user is also informed when the game has been lost, this is done by opening all mines (the one which was clicked is highlighted red) and changing the icon of the reset button to be dead. This functionality is shown in **Test 11**. This objective could be subject to the same improvements as **Objective 7**.

8. **When the reset button is pressed, a new randomly generated board with the same dimensions and same number of mines should be created for the user to play.**

   This objective has been met, as shown in **Test 6**. The reset button also causes the timer to reset. This objective doesn't have a clear way to improve upon.

9. **The speed of the Minesweeper solver should be such that there are no breaks for computation longer than 5 seconds that inconvenience the user of the program.**

   The solver has some breaks for computation, the frequency of which correlates to the difficulty of the board. On an Intermediate board (as shown in **Test 12**) there is almost no breaks, on an Expert board there are very few breaks longer than 5 seconds. This is the objective for which most improvement can be made. Due to the time that more complex positions take to solve, I had to make some compromises to accuracy to ensure the speed remains bearable. Solvers exist which are more efficient than mine and can solve more complex positions quicker and more accurately (using global guessing). If I were to redo this project, I would prioritise the accuracy of my program, which would require quicker algorithms that can consider solutions more efficiently.

10. **The Minesweeper solver should have a win rate of between 25 and 30% when measured over a sample of 10,000 expert boards.**

    This was, for me, the most important objective, although my end user was not overly interested in maximising the solver's win rate. In the end I managed to achieve a win rate of around 29% in Expert (shown in **Test 13**), which is comfortably within my desired range. This is another objective which has a very high ceiling for improvement which, given the time, I would've liked to improve more upon.

11. **After every game, the stats for that game (including difficulty, result, time etc) should be written to a human readable stats file contained within the programs resource folder.**

    I am happy with the way I store stats for the games which is shown in **Test 14**, and especially with the tailor-made format for storing boards to be recreated. My main area for improvement on this objective, however, is with the size of the stats file. When many games have been played, this file can become very large, if I were to redo this project, I

would consider using some form of compression method to decrease the size, although this compression would have to be lossless to preserve the exact stats.

12. **A visual display of the stats for every game played by both human and computer using the program should be available for browsing.**

    My stats table is fairly well designed in my opinion, it contains relevant information in an easy-to-understand format. To improve my visual display, a better algorithm for reading in the data could be used. This improvement would go hand in hand for the improvement for **Objective 11** and would mean that a large number of games could be read and displayed without the short, but noticeable time delay that occurs currently. Another improvement that could be made is the inclusion of more information, such as an overall win rate or an average time, these were not originally included as the variety in games in the file would make such information relatively useless although it may be useful to an end user.

13. **These stats should be filterable and sortable by Difficulty, Result, Time and 3BV.**

    All the filters are available to use, and sorting is a built-in function of DataTables in VB. The only improvement to this I can think of would be a more seamless integration of the Less Than feature in the numerical filters, as the checkbox isn't the most intuitive, this would be very low priority if I were to redo this program though.

14. **A live tracker of the results of the solver, including a running tally of games and the current win rate, as it is playing the game should be available for the user when the computer is playing the game.**

    The Live Stats tracker is fully functional as shown in **Test 15**. Currently, it only tracks when the computer is playing. A future iteration of this program could extend this to show the user a live tracker as they are playing, although the information about guesses would have to be removed or modified to be relevant to the humans experience.

15. **The user should be able to create any Minesweeper board up to and including a 50x50 grid, either by manually inputting values or using an image.**

    Custom Boards can be made up to and including 50x50 as evidenced in **Tests 16-18**. The custom board feature has one obvious improvement which could be made, adding an (optional) mine count. This would allow the user to gain more accurate information about a specific board using global guessing. My reasoning for not including this feature was that my program was that I designed this feature to give the "general solution" to a position (or the local guessing solution) as it would be relevant, if a little imprecise, for any number of mines.

16. **The computer should detect if a user generated grid is valid, and if so, correctly determine the locations of any mines, safe cells, or best guesses and visually display that data to the user by shading the cells on the board with different colours.**

    As referenced in the evaluation for **Objective 15**, my program uses local guessing to find solutions to a user generated board. This works as intended, and the shaded cells make this an intuitive and easy to use feature for the user as demonstrated in **Test 19**, it also successfully detects invalid boards as shown in **Test 20**. Improving upon this would involve

the same process detailed in the above evaluation, where a mine count is supplied and global guessing can be utilised.

17. **The user should have the ability to save an image of any Minesweeper board they have created directly to their computer using the computers built in file explorer.**

    The final required objective works as intended, with one caveat. When an image is being copied from an image to the program, the board will only be recognised if each cell is of a certain size (16px), as that is the size of a cell in the program. This works flawlessly for any boards generated by my program (using the custom board feature or from an actual game) and can also find a board within an image containing other information, such as a screenshot. This, however, means that a board taken from a different implementation of the game is unlikely to be recognised. This isn't a huge issue as the board can be inputted manually, and is one which would be extremely difficult to remedy due to the visual differences in different versions of the game. This would not be a high priority to fix if this project was redone.

Optional Objectives

1. **The user should be able to choose any game from stats, and have that board recreated for playing again.**

   This feature was implemented by using the stats file and storing the board there. It is a useful option to have, although is the main reason that the stats file gets so large as more games are played. Because of this, and the fact that this objective is optional, this would not be a necessity if I was to redo this project.

2. **Give the user the option to choose different difficulties of boards and generate a board of that difficulty which can be played by either human or computer.**

   This objective was met through the training function. It allows you to choose a difficulty of board based on algorithms used by the computer. There are many ways to improve upon this training feature as it (as far as I know) has never been done for Minesweeper before. A couple of improvements could be a way of determining a difficulty of a board without playing the entire game (similar to 3BV) although this may be an impossible task, alternatively a guide on how to solve a board if the user was stuck, which would be a useful feature but well beyond the requirements of the end user to my program. These extra features could be considered for a future version of this program.

Final Evaluation with End User

Overall, how happy are you with my final program?

The program does everything I asked of it and the interface is easy enough to use, so overall I would say that I'm satisfied with the program.

Do you have any comments about the solver?

The solver feels easy to use, especially with the added feature of a delay so I can see the individual moves it makes more easily. If I were to suggest an improvement to the solver it

would be to improve its speed, sometimes it does have some longer breaks that can be slightly frustrating. Overall, however, I am happy with the ease of use and accuracy of the solver.

Did you find the Live Stats and Overall Stats function useful?

How about the Custom Boards?

The custom boards are my favourite part of the program as they allow me to analyse and find the solutions to positions, I wouldn't usually be able to. The shading makes it very easy to understand, and the controls are intuitive and efficient. I have noticed that sometimes, the probability of a guess is different to one given by a different program however, which is confusing.

Is the ability to return to a previously played board useful to you?

This feature is also really appreciated, sometimes I make mistakes when I play, and want to retry that board again. It also helps when I get stuck and want the computer to play through the board and find where I went wrong. The training function is nice but feels incomplete.

Do you feel like the initial objectives have been met?

On the whole, I believe that the initial objectives have been achieved, and in some cases, the program goes further than I would've expected. This is appreciated when it comes to features such as the custom boards and stats, although I feel like the solver could've been made better if more time was allocated to improving it. Having said that, the program is exactly what I wanted when requesting this project, and does everything I asked of it.

Final Comments

Having spoken to my end user, I believe he agrees with my evaluation of the objectives. I am especially happy with the positive feedback for the Custom Boards, as a lot of effort was put into its ease of use. We both agree that the stats function is useful in its current form, although could be extended upon given more time to include extra information.

The training function was slightly rushed, which reflects in Chris' comment about it being "incomplete". This is an idea which could make for an interesting and very useful project at a later date, as it has so many possibilities which I didn't have time to explore in this project.

The solver itself is the main part of the program, and I feel that its final form is satisfactory to a user who just wants to watch the computer play Minesweeper and possibly learn some patterns from it. Where the solver fails somewhat is for more complex analysis of boards, Chris commented on the solver taking a long time for computation for some boards, which was a point I also made in my own evaluation. This comes down to how the logic is performed, and given more time I would most likely attempt a complete redesign of the structure and logic of the solver in an attempt to increase its efficiency, which would in turn allow it to accurately solve more complex positions.

References

[1] – Minesweeper Rankings - https://minesweeper.online/ranking

[2] – Minesweeper Reddit - https://www.reddit.com/r/Minesweeper/

[3] – Calculating 3BV - https://gamedev.stackexchange.com/a/63048

[4] – 3BV Limits on Trial, Ronny de Winter, October 17th 2009 - http://www.minesweeper.info/articles/3BV_Limits_On_Trial.pdf

[5] – 1-2 Pattern - https://minesweeper.online/help/patterns#1-2

[6] – MrGris Minesweeper Analyser - https://mrgris.com/projects/minesweepr/demo/analyzer/

[7] – Introduction to NP-Completeness - https://www.geeksforgeeks.org/introduction-to-np-completeness/

[8] – DTM's vs NDTM's - https://stackoverflow.com/a/127831

[9] Kaye, R, Minesweeper is NP-complete, *The Mathematical Intelligencer* **22**, 9–15 (2000) - https://doi.org/10.1007/BF03025367

[10] – P vs NP Millennium Prize Problem - https://www.businessinsider.com/p-vs-np-millennium-prize-problems-2014-9?r=US&IR=T

[11] – Minesweeper: Advanced Tactics - https://nothings.org/games/minesweeper/

[12] – Minesweeper as a Constraint Satisfaction Problem, Chris Studholme - https://www.cs.toronto.edu/~cvs/minesweeper/minesweeper.pdf

[13] - .NET Filter Expression Syntax - https://learn.microsoft.com/en-us/dotnet/api/system.data.datacolumn.expression?view=net-7.0

Videos

[1] – Chording Example - https://youtube.com/shorts/EGWF8V3AgzM

[2] – Blank Cells Example - https://youtube.com/shorts/NEllsYiUxVg

[3] – Live Stats Display - https://youtu.be/WVhjNW2iQUM

[4] – Custom Board Demonstration - https://youtu.be/HNpn0NzOqc4