Linniel Driver-Williams

B420810

Minesweeper Learner Using Case-Based Systems

Final Year Project | Loughborough University

2016-2017

# Abstract

The game of Minesweeper has a simple predefined set of rules which while easy enough to pick up give rise to a whole host of puzzling board states and challenges.

These emergent puzzles require strategy, logical reasoning and a sprinkling of luck to solve. And it is because of this that humans pit their wits against the board; navigating through a field of mines all the while trying to keep their feet.

Given its simple rule set, it is rather simple to implement a Minesweeper ‘solver’, a program which given a rule set solves the puzzle which in this case is the minefield or board. Given it knows the distinct rules of the game a sweeper learner can just as easy create emergent moves to solve the emergent puzzles.

While solvers exist for Minesweeper the question comes around: Can we build an AI that learns how to play Minesweeper to a reasonable level? It may not make sense to create a learner for something we have a solver for, creating such an AI in an environment where we know the outcome may be useful in order to practice and learn ideas for more practical uses such as in a water treatment plant or in protein folding.

In this paper I outline my adventures into the world of AI in regards to teaching my computer how to play Minesweeper! I’ll be discussing Minesweeper, learning principles and the obstacles I had to overcome to create my learner.

# Keywords

Case-Based Reasoning, Artificial Intelligence, Minesweeper, Database

# Acknowledgements

I would like to express my profound respect and appreciation for the people who made it possible to complete my report. To my final year project supervisor Dr. D.Reidenbach, who’s intellectual stimulation and support gave me constant feedback and ideas to include in my project.

And also to my friends who I also was able to bounce ideas off, discuss concepts with and to use as my informal test subjects, without which this reports comparisons would be a lot weaker.

Table of Contents

[Abstract 1](#_Toc481562482)

[Keywords 1](#_Toc481562483)

[List of Figures & Tables 3](#_Toc481562484)

[1 Introduction 5](#_Toc481562485)

[2 Literature Review 6](#_Toc481562486)

[3 Requirements 7](#_Toc481562487)

[4 Methodology 8](#_Toc481562488)

[5 Design 10](#_Toc481562489)

[6 Case Design (Numbering and Size) 13](#_Toc481562490)

[7 Implementation 15](#_Toc481562491)

[7.1 Code Outline and Extracts 15](#_Toc481562492)

[7.2 Implementation Problems 22](#_Toc481562493)

[7.2.1 Database Integration & Limitations 22](#_Toc481562494)

[7.2.2 Database Optimisation 22](#_Toc481562495)

[8 Testing 23](#_Toc481562496)

[8.1 Minor Adaptation Testing 23](#_Toc481562497)

[8.2 Testing Theory 24](#_Toc481562498)

[8.3 Analysis of results 24](#_Toc481562499)

[8.3.1 Forgetful CBR 24](#_Toc481562500)

[8.3.2 Traditional CBR 27](#_Toc481562501)

[8.3.3 Edge Detection CBR 28](#_Toc481562502)

[8.4 Other Comparisons 33](#_Toc481562503)

[9 Specific Case Analyses 34](#_Toc481562504)

[10 Conclusion 35](#_Toc481562505)

[11 References 37](#_Toc481562506)

# List of Figures & Tables

[Figure 1‑1: Board populated with mines 5](file:///C:\Users\Linniel\Documents\_FYP%20Spec\FYP%20Proper.docx#_Toc481562507)

[Figure 1‑2: Board populated with mines and clues 5](file:///C:\Users\Linniel\Documents\_FYP%20Spec\FYP%20Proper.docx#_Toc481562508)

[Figure 1‑3: Blank minesweeper board 5](file:///C:\Users\Linniel\Documents\_FYP%20Spec\FYP%20Proper.docx#_Toc481562509)

[Figure 4‑1: The Case-Based Reasoning Cycle 8](file:///C:\Users\Linniel\Documents\_FYP%20Spec\FYP%20Proper.docx#_Toc481562510)

[Figure 4‑2: A cell with scoring cells highlighted in green 8](file:///C:\Users\Linniel\Documents\_FYP%20Spec\FYP%20Proper.docx#_Toc481562511)

[Figure 5‑1: A flowchart of the main application cycle 10](file:///C:\Users\Linniel\Documents\_FYP%20Spec\FYP%20Proper.docx#_Toc481562512)

[Figure 5‑2: A flowchart of the primary game cycle, not including the zero-flush subroutine 10](file:///C:\Users\Linniel\Documents\_FYP%20Spec\FYP%20Proper.docx#_Toc481562513)

[Figure 5‑3: My original CBR Deciding and Learning algorithms 11](file:///C:\Users\Linniel\Documents\_FYP%20Spec\FYP%20Proper.docx#_Toc481562514)

[Figure 5‑4: A UML Class Diagram of all my classes used 12](#_Toc481562515)

[Figure 6‑1: A cell of undetermined value (highlighted in red) and two regions of interest with varying diameters 13](file:///C:\Users\Linniel\Documents\_FYP%20Spec\FYP%20Proper.docx#_Toc481562516)

[Figure 6‑2: Two different cells of undetermined value, regions of interest and a zone of interest which a normal human might use 13](file:///C:\Users\Linniel\Documents\_FYP%20Spec\FYP%20Proper.docx#_Toc481562517)

[Figure 6‑3: A log. graph visually showing the size of each casebase 14](#_Toc481562518)

[Figure 6‑4: A 3\*5 case examples 14](#_Toc481562519)

[Figure 7‑1: Minesweeper board initialisation code snippet 15](#_Toc481562520)

[Figure 7‑2: A 5\*5 sample with a 5\*3 case/region of interest indicated by the dotted line with a cell of interest in the corner of the board indicated by the thick black border, green cells being part of the case, grey cells being part of the case but will always be null values and cyan cells which are part of the board but not part of the case region. 15](#_Toc481562521)

[Figure 7‑3: Mine and clue generation code snippet 16](#_Toc481562522)

[Figure 7‑4: Cell reveal and zero-flush code snippet 16](#_Toc481562523)

[Figure 7‑5: An example of a compression from a subset of a 3-dimesional to a 2-dimensional array with the cell of interest on the bottom row 17](#_Toc481562524)

[Figure 7‑6: The final Arraylists used as the CBR's casebase 17](#_Toc481562525)

[Figure 7‑7: The 4 regions of interest checked by my CBR. 18](#_Toc481562526)

[Figure 7‑8: A code snippet of the CBR searching the casebase for a match 18](#_Toc481562527)

[Figure 7‑9: A code snippet showing the additions made to the CBRs decision process to factor in edge scores 19](#_Toc481562528)

[Figure 7‑10: A code snippet that shows how the CBR adding new cases to the casebase 19](#_Toc481562529)

[Figure 7‑11: A code snippet showing what happens after a case is found and reveals a safe cell 20](#_Toc481562530)

[Figure 7‑12: The code which the CBR uses to 'forget' a case 20](#_Toc481562531)

[Figure 7‑13: The code which increments the usage statistics after a mine is revealed 20](#_Toc481562532)

[Figure 7‑14: A code snippet of the save process 21](#_Toc481562533)

[Figure 7‑15: Profiler showing how objectdb functions require the majority of my computation time 22](#_Toc481562534)

[Figure 7‑16: A snapshot of the playAI method computation breakdown 22](#_Toc481562535)

[Figure 8‑1: A beginner Minesweeper board with the shaded cells being areas where there would be an out of bounds token 23](#_Toc481562536)

[Figure 8‑2: A graph showing how the number of wins my forgetful CBR achieves over the amount of games played 24](#_Toc481562537)

[Figure 8‑3: A graph showing how much of the casebase my forgetful CBR actually retained 25](#_Toc481562538)

[Figure 8‑4: A graph showing my forgetful CBRs casebase size over the amount of games played 25](#_Toc481562539)

[Figure 8‑5: A graph showing how the number of wins my forgetful CBR with both adaptations achieves over the amount of games played 26](#_Toc481562540)

[Figure 8‑6: A graph showing how the number of wins my forgetful CBR with the win-learning adaptation achieves over the amount of games played 26](#_Toc481562541)

[Figure 8‑7: A graph that shows the size of the casebase as the amount of games my forgetful CBR plays 27](#_Toc481562542)

[Figure 8‑8: A graph showing how the number of wins my traditional CBR with the win-learning adaptation achieves over the amount of games played 27](#_Toc481562543)

[Figure 8‑9: A graph showing how the number of wins my traditional CBR with the win-learning adaptation achieves over the amount of games played with a highlighted area of interesting growth 28](#_Toc481562544)

[Figure 8‑10: A graph showing how the number of wins my edge-detection CBR with the win-learning adaptation achieves over the amount of games played 29](#_Toc481562545)

[Figure 8‑11: A graph that shows the percentage of Non-random clicks made per 1000 games 29](#_Toc481562546)

[Figure 8‑12: A graph that shows how my Edge-Detection CBRs casebase grows as the CBR played 30](#_Toc481562547)

[Figure 8‑13: A graph showing how the number of wins my edge-detection CBR with the win-learning adaptation achieves over the amount of games played for its first 146 thousand games 30](#_Toc481562548)

[Figure 8‑14: A graph which shows the trend line of both my Traditional and Edge-Detection CBRs modelled as 6th power polynomials 31](#_Toc481562549)

[Figure 8‑15: My original edge-detection CBR (with win-learning) and its results for the 100 hours proceeding 32](#_Toc481562550)

[Figure 8‑16: My edge-detection CBR (with win-learning) with a 66% selection threshold 32](#_Toc481562551)

[Figure 8‑17: A bar chart showing win-rates of the player groups, solver and learners 33](#_Toc481562552)

[Figure 8‑18: The two Intermediate games my Edge-Detection CBR with win-learning won 34](#_Toc481562553)

[Figure 9‑1: The most used case in my edge detection CBR 34](#_Toc481562554)

[Figure 9‑2: The third most used case in my edge detection CBR 34](#_Toc481562555)

[Figure 9‑3: Cases 149442 and 18319 respectively, the only two cases in the top 50 most used that were not created in the first 10 thousand games 35](#_Toc481562556)

# Introduction

In this project I’ve been tasked with implementing one or more algorithms that learn how to play the game ‘Minesweeper’. To this end, due to the fact I could not find a Minesweeper implementation which suited my needs, I implemented my own version of Minesweeper.

As a preface Minesweeper is a single-player game created by Curt Johnson and is often packaged with operating systems. The player is given a p \* q board with n mines randomly distributed on it (see figure 1-1). The squares remaining are then assigned a number given how many mines are adjacent or diagonal to them (see figure 1-2).

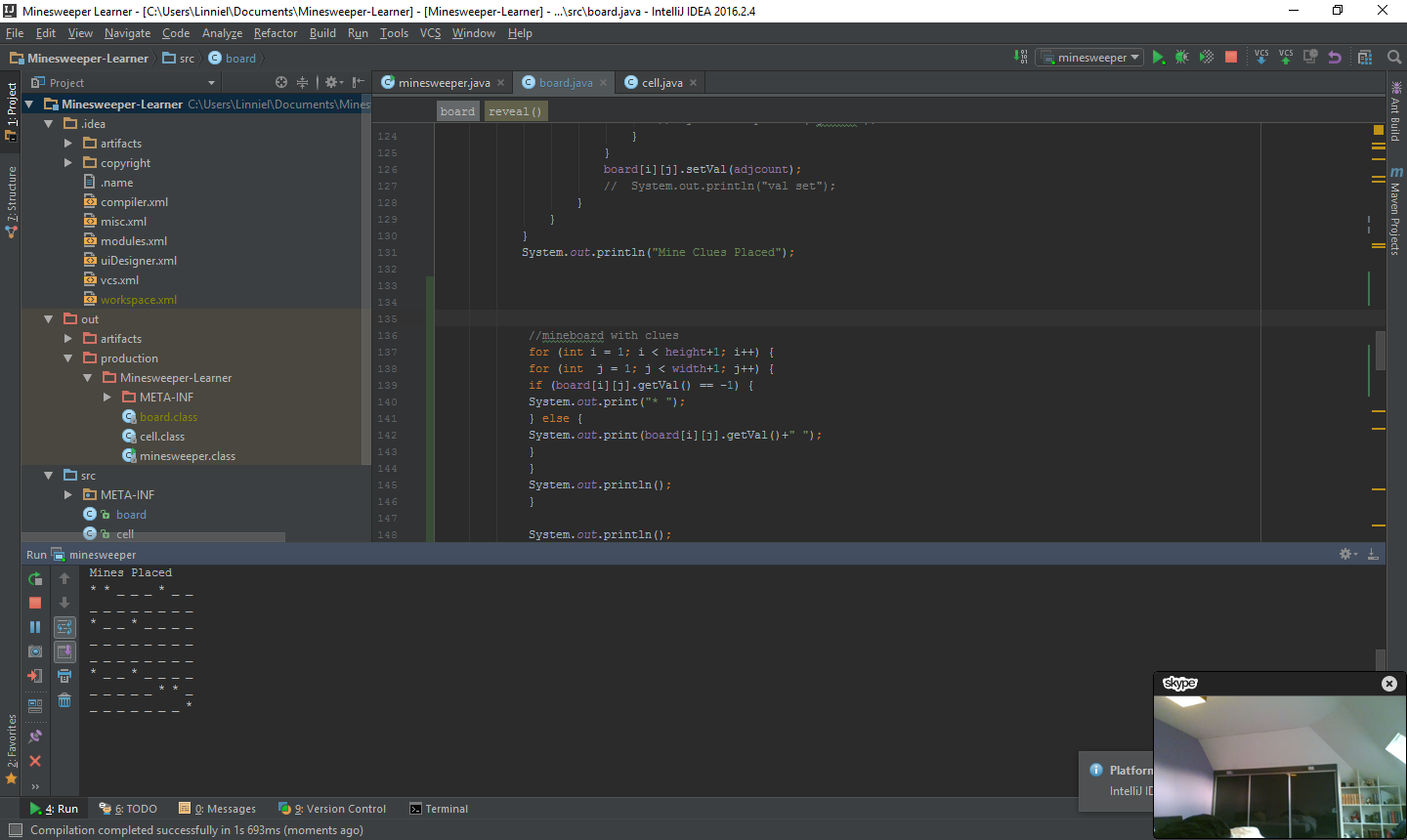
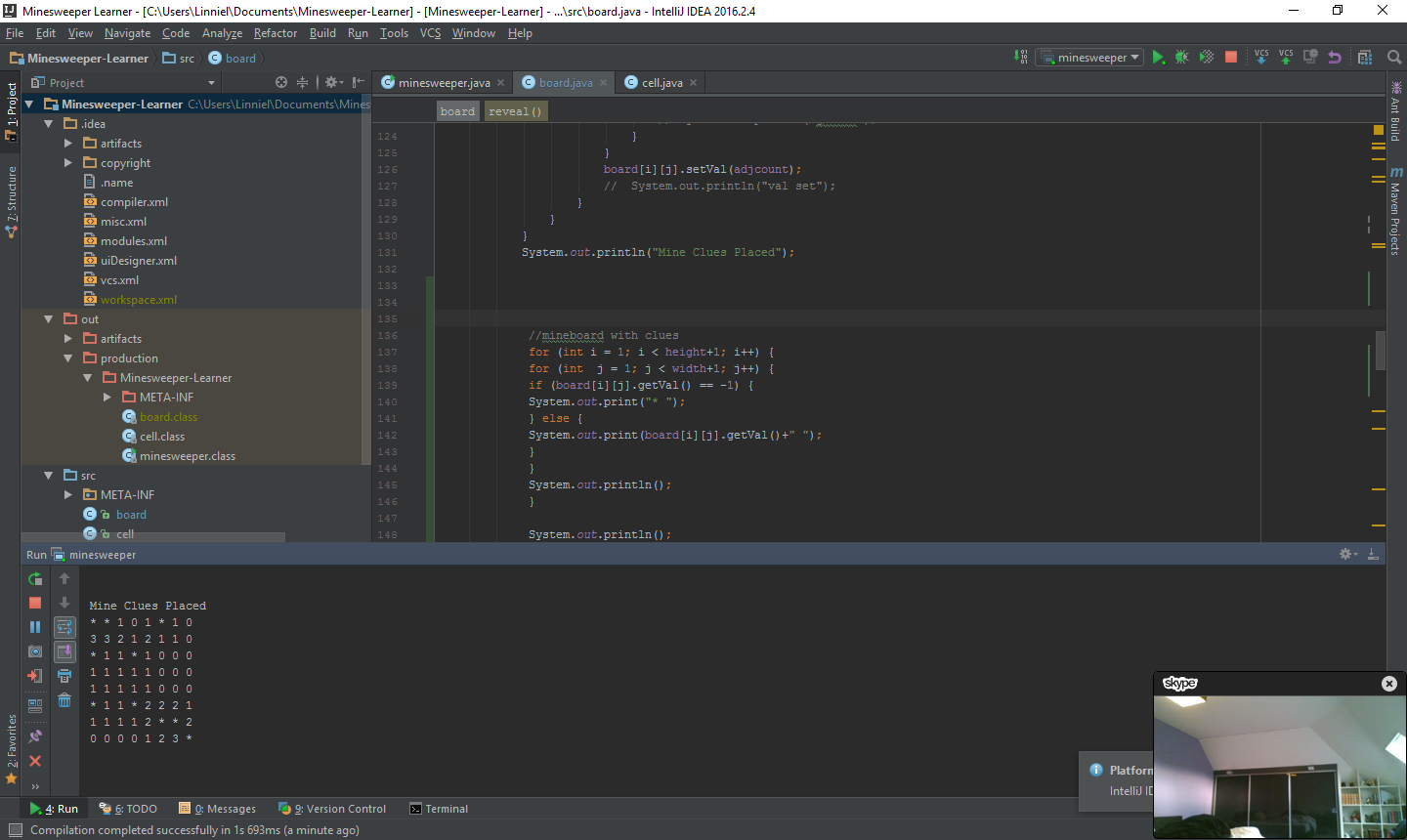


Figure 1‑1: Board populated with mines

Figure 1‑2: Board populated with mines and clues

At the start of the game, the board is completely blank and the mines are hidden to the user (see figure 1-3). From here, using the clues provided by the game board as cells are revealed, the player has to reveal all the non-mine cells on the board without activating a mine cell to win the game.

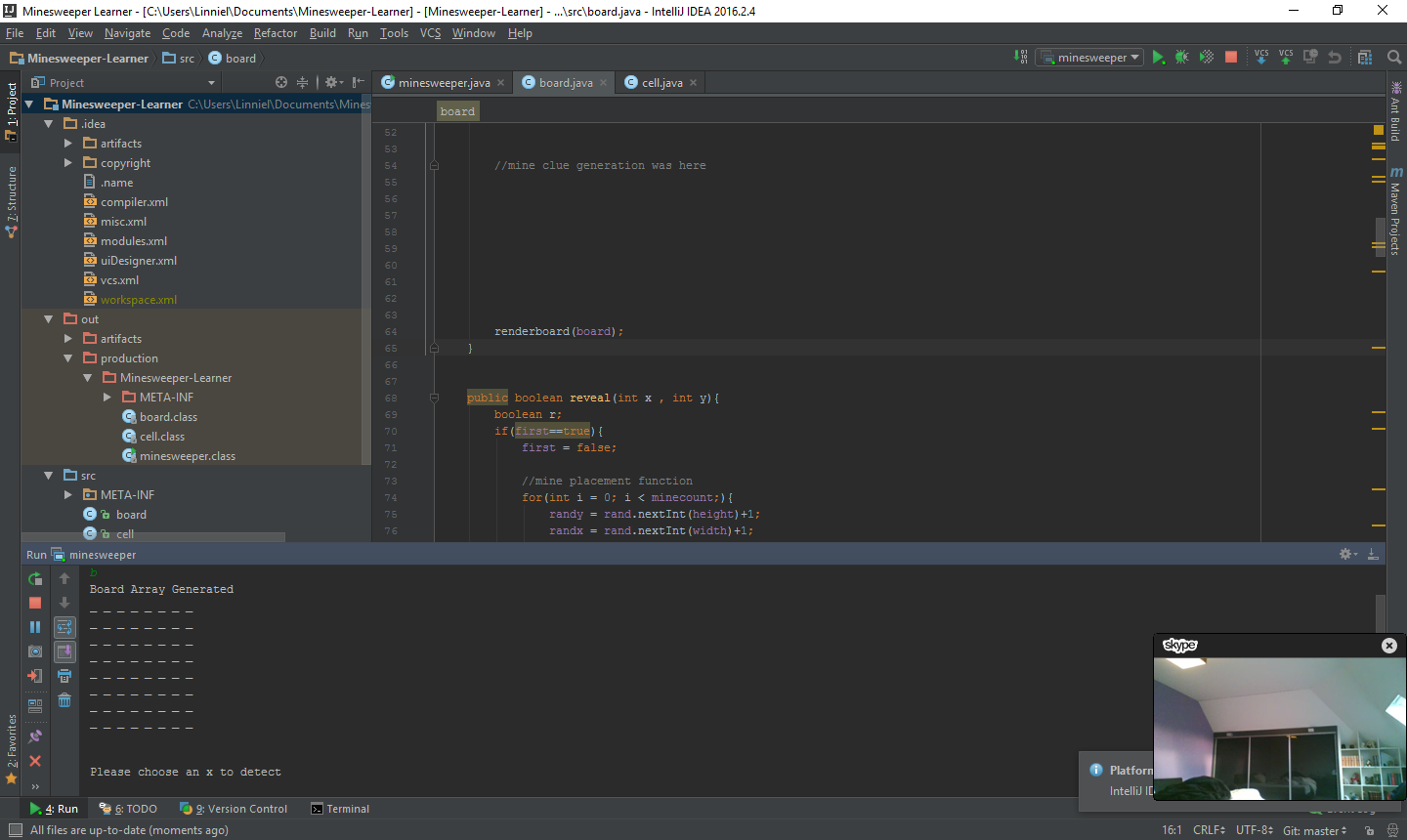


Figure 1‑3: Blank minesweeper board

My Minesweeper algorithm is a ‘learning algorithm’ which, as the name suggests, learns how to play Minesweeper by itself by self-improvement over time rather than a ‘solver’ which has a predefined rule-set built in to solve the minefield (the game’s board).

The algorithm model which I based my algorithm model on is the ‘Case-Based-Reasoning’ (CBR) method as we know the actual rules behind Minesweeper but has a large number of potential cases which we can draw from. The ‘run of a mill’ Minesweeper CBR also has a series of adaptations which allowed for some interesting comparisons between data. These experimental results show us how the CBR functions, its pros, cons and limitations as compared to that of a human, an AI solver and other AI learners.

# Literature Review

There have been many attempts at creating learners for Minesweeper, but none seem to have used a Case-Based Reasoning approach. In a paper by Kaye (2000) he outlined the fact that Minesweeper is in fact NP-Complete by simulating Boolean circuitry as Minesweeper boards. As an NP-complete problem, there is no known algorithm to solve it in polynomial time and such we have to resort to using smarter tactics rather than pushing through and brute-forcing a solution.

There are two ways we can do this, one could implement a rule set for a Minesweeper solving program, a solver, or one could implement a program which teaches itself how to play Minesweeper, a learner.

Roos (2015) implemented an extremely potent solver which utilised rules and controlled risk in order to reveal board information and was able to achieve an impressive 81.1% win-rate on a classic beginner board with an edge and corner cell selection priority.

Over the years there have been many attempts to get a computer to approximate or induct the game of Minesweeper’s rule-set in order to reveal non-mine cells. One such example used by Castillo and Wrobel (2003) made use of multi-relational learning to create a series of rules for their AI to follow to both detect mine and non-mine cells and achieved a respectable win-rate of 57.9%. Gardea et al (2015) used Q-learning to limited success solving 4\*4 boards with startling success rates but unfortunately was not an expandable solution although they did manage to sustain high board rates by framing Minesweeper as a Constraint Satisfaction Problem. While both Rhee (2000) and Quartetti (1998) used genetic algorithms to solve boards.

Despite these many attempts, there is still much work and methodologies which could be applied to Minesweeper learners. Very little work has been done in neural networks and next to no discoverable work in case based systems. Because of this there is an exciting gap in research which could be further explored such as evolving neural networks with evolving topologies as described by Stanley and Miikkulainen (2002) or a pure case based reasoning approach, the latter of which I shall be focusing on.

Case based reasoning as outlined by Schank (1983) is a way of outlining rules by a collective experience earned over many iterations, a process that was inspired by human reasoning and memory. In the same way a human can recall a memory, a CBR acts as a hyper-heuristic which recalls an event and can respond by calling upon these previous memories as a template that allows it to recount and adapt its actions and depending on the outcome, results in this action being reinforced or weakened. It is this reinforcement of cases that allows the CBR to develop its behaviours.

An advantage CBRs have over other static learning methods is that as the casebase grows, finer intricacies and fringe cases are remembered, something which may be a lot harder to define by something like a static neural network or through traditional statistical reasoning. This, in theory, given a complete enough casebase, should result in a higher win-rate overall and is the hypothesis behind my project.

# Requirements

In order to accomplish the goals outlined earlier I implemented a program with the following requirements:

1. Minesweeper Game
   1. Must have a playable game of Minesweeper which is controllable by both humans and my AI which generates a board of a specified size with a specified number of mines and creates a series of clues to guide the player, allowing them to win or lose.
   2. Must not lose on the first move. (While a good idea to reduce redundant game cycles, created a slight oversight later).
   3. Have a ‘zero-flush’ feature, where upon clicking a zero, all the other cells which are next to that zero are also revealed. This is to speed up play time and also to reduce the number of cases in a case based system.
2. Case-Based Reasoning Learner
   1. Must be able to decide on which cell on the board to reveal based on a given criteria and/or metric to bring the board to a solved state.
   2. Must be able to ‘learn’ from past experiences through a database of previously seen ‘cases’.
      1. Must be able to store new cases when the machine sees something new.
      2. Must be able to overhaul and recalculate previously stored memories with newly acquired information.
3. Case-Based Reasoning Learner Adaptations
   1. To implement a ‘Forgetful Adaptation’ which allows the computer to delete cases from its database which have led the AI to reveal a cell containing a mine, preserving the ‘purity’ of the database.
   2. To implement an ‘Edge Detection Adaptation’ which alters the decision cycle to take into account revealed ‘edges’ making the AI utilise more board information in the hopes of making more informed decisions.

# Methodology

This project will be utilising case-based reasoning machine learning in order to build a Minesweeper AI which constantly betters itself.

Figure 4-1 (Aamodt and Plaza, n.d) outlines case based reasoning into four different steps:

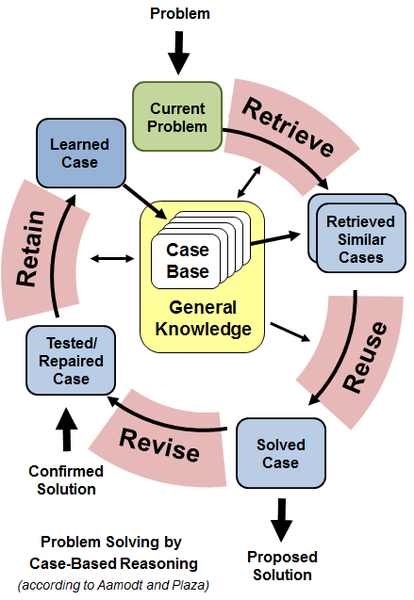


Figure 4‑1: The Case-Based Reasoning Cycle

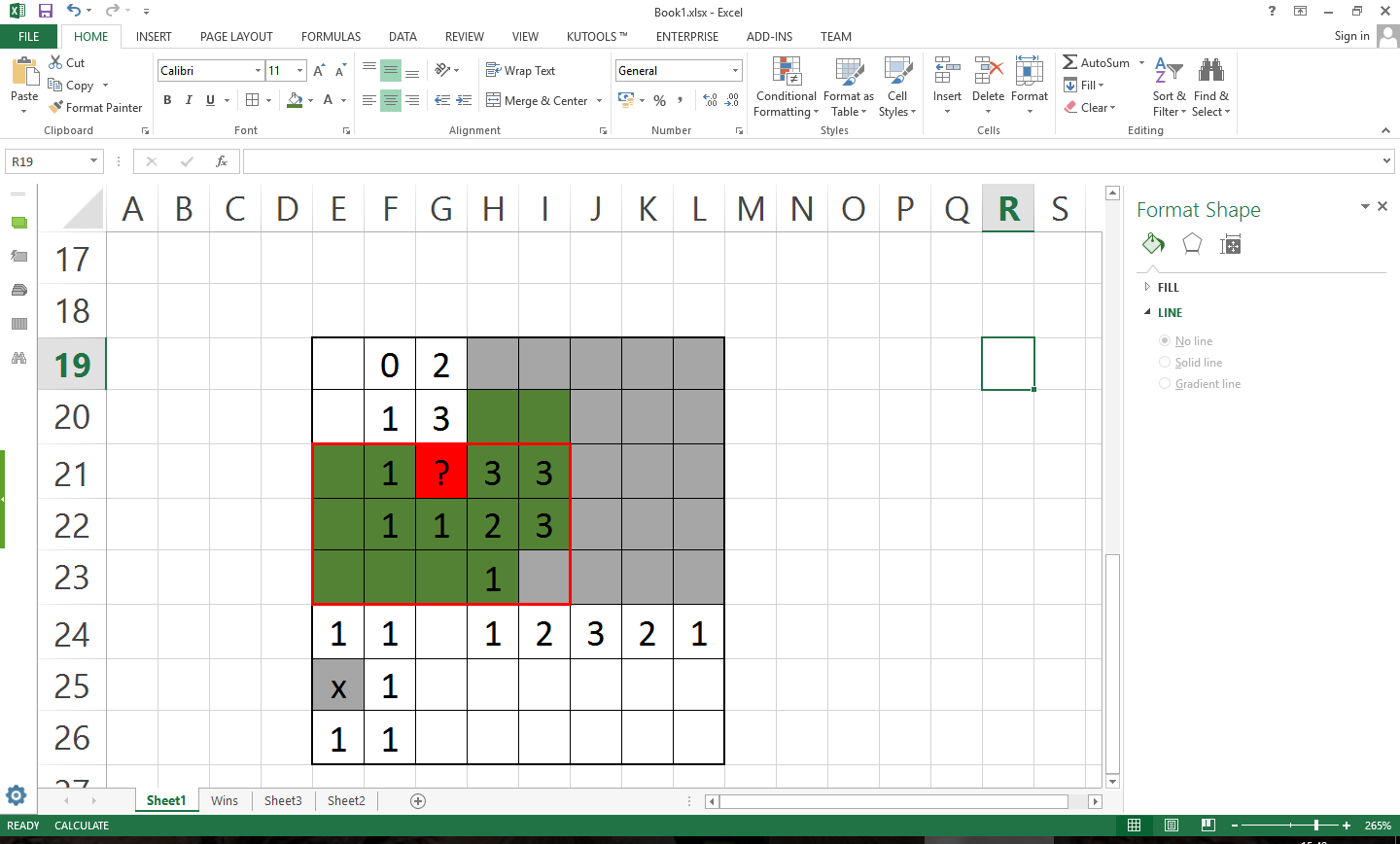
* Retrieve
* Reuse
* Revise
* Retain

My CBRs will cycle through the Minesweeper boards, *retrieving* sub-boards to compare against the casebase. Once it has found a solution to use it then *reuses* this case. Upon finding out the outcome it then *revises* that case’s statistics, but does not physically alter the case’s template. After that it then *retains* this new information and/or new case to the casebase and the process starts over.

As outlined in my requirements, I created three separate Minesweeper CBR implementations along with minor variations on each of these. These are:

1. Pure ‘Vanilla’ Case Based Reasoning Learner
2. ‘Forgetful’ Case Based Reasoning Learner
3. ‘Edge Detection’ Case Based Reasoning Learner

The first is a case-based system in the purest of senses; upon being presented a board it gathers all the cases on the board and scans the casebase for a match, selects the case with the highest win-rate and learns from its successes and failures, finding cases that it deems worthy of using with no other pieces of injected information.

The second is an extremely strict CBR version that erases the case in its entirety if they lead the sweeper to click on mine cell, with the theory that by keeping the case-base small it reduces the time to complete games thus increasing the learning rate while keeping a concentrated casebase. With the hope that eventually the casebase would reach a ‘critical mass’, not having to learn many new cases anymore and have a pure casebase that can handle most puzzles proposed to it. As such it became important to have an optimised region of interest for my CBR to inspect such that it can gather the necessary information to win games but also small enough to hit this theoretical ‘critical mass’.

The last method, which was realised and developed after case design, utilises a scoring system to prefer cells with a higher ‘edge score’ (with an acceptable win-rate) over those which may have a higher win-rate. The edge score being the sum of all revealed cells in the region of interest plus the number of unrevealed cells along the column or row next to the cell being inspected, as seen in figure 4-2. For example; if a cell had an edge-score of 16 and a win-rate of 87% and another cell had an edge score of 9 but a win-rate of 90%, this version of the CBR would pick the former cell over the latter.

Figure 4‑2: A cell with scoring cells highlighted in green

This is to replicate how humans play Minesweeper, by flowering out from a central position, revealing cells at an edge, using the board information it has already gathered, like from a zero-flush which human players work their way around, rather than from what essentially amounts to a ‘gut feeling’, stabbing away at more token pieces of information.

# Design

Figure 5-1 outlines the main body of the program and how it operates, it is a simple program which was kept neat and clean to allow easy testing and modularity, easing the burden of both unit testing and integration.

Figure 5‑1: A flowchart of the main application cycle

In figure 5-2 I outline the general principle behind the primary minesweeper implementation. While it was originally implemented with a human player in mind, its construction also accounted for an AI player, thus satisfying my requirement’s first point.

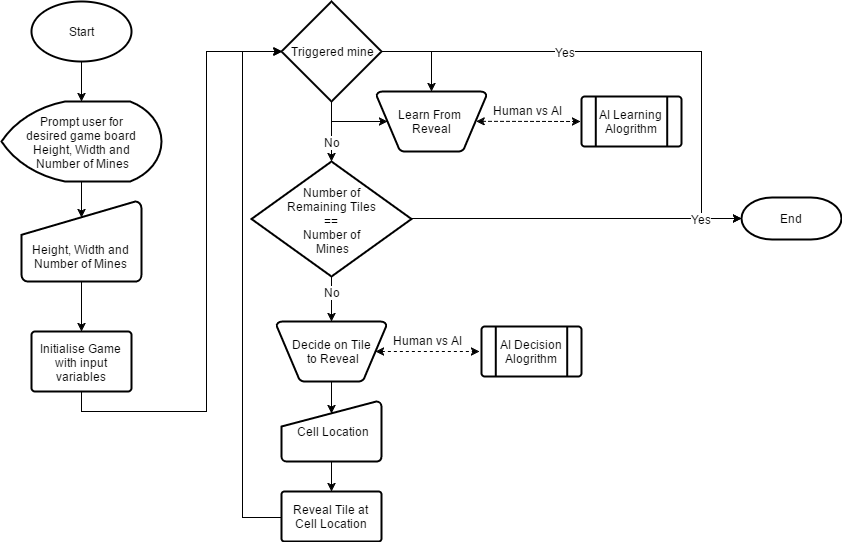
In figure 5-3 I have described my original AI decision and learning pipelines for my forgetful CBR which I later changed due to database optimisation issues.

Figure 5‑2: A flowchart of the primary game cycle, not including the zero-flush subroutine

And in figure 5-4 we can see a class diagram of my program to give an outline of my implementation.

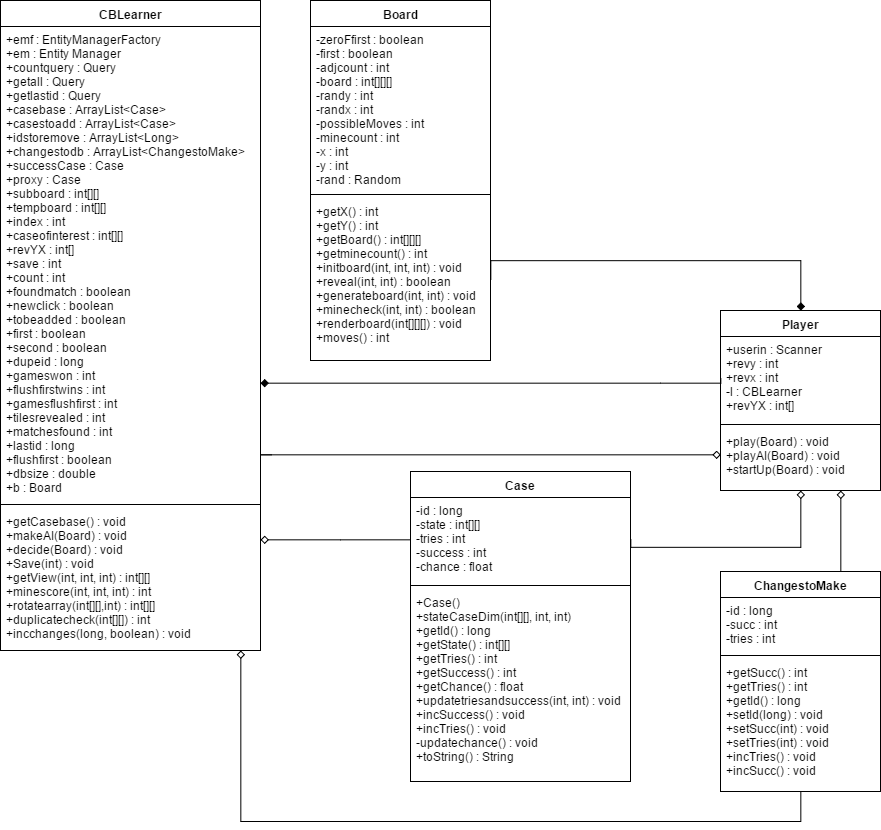
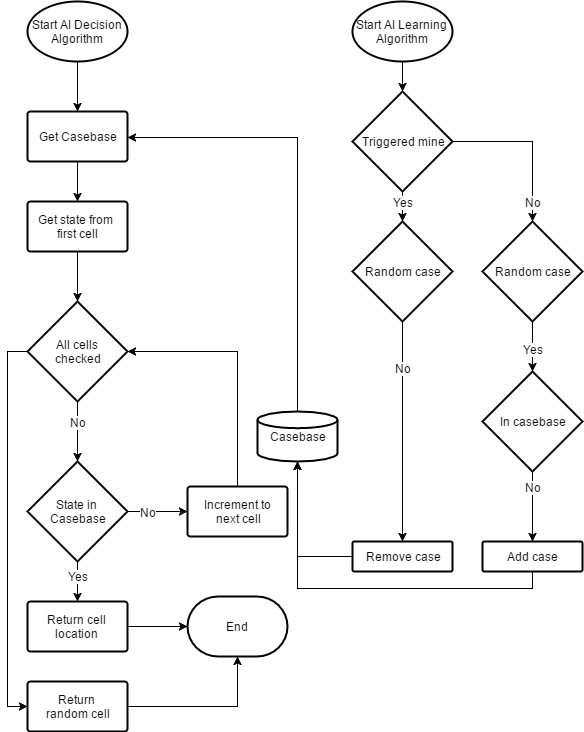


Figure 5‑3: My original CBR Deciding and Learning algorithms

Figure 5‑4: A UML Class Diagram of all my classes used

# Case Design (Numbering and Size)

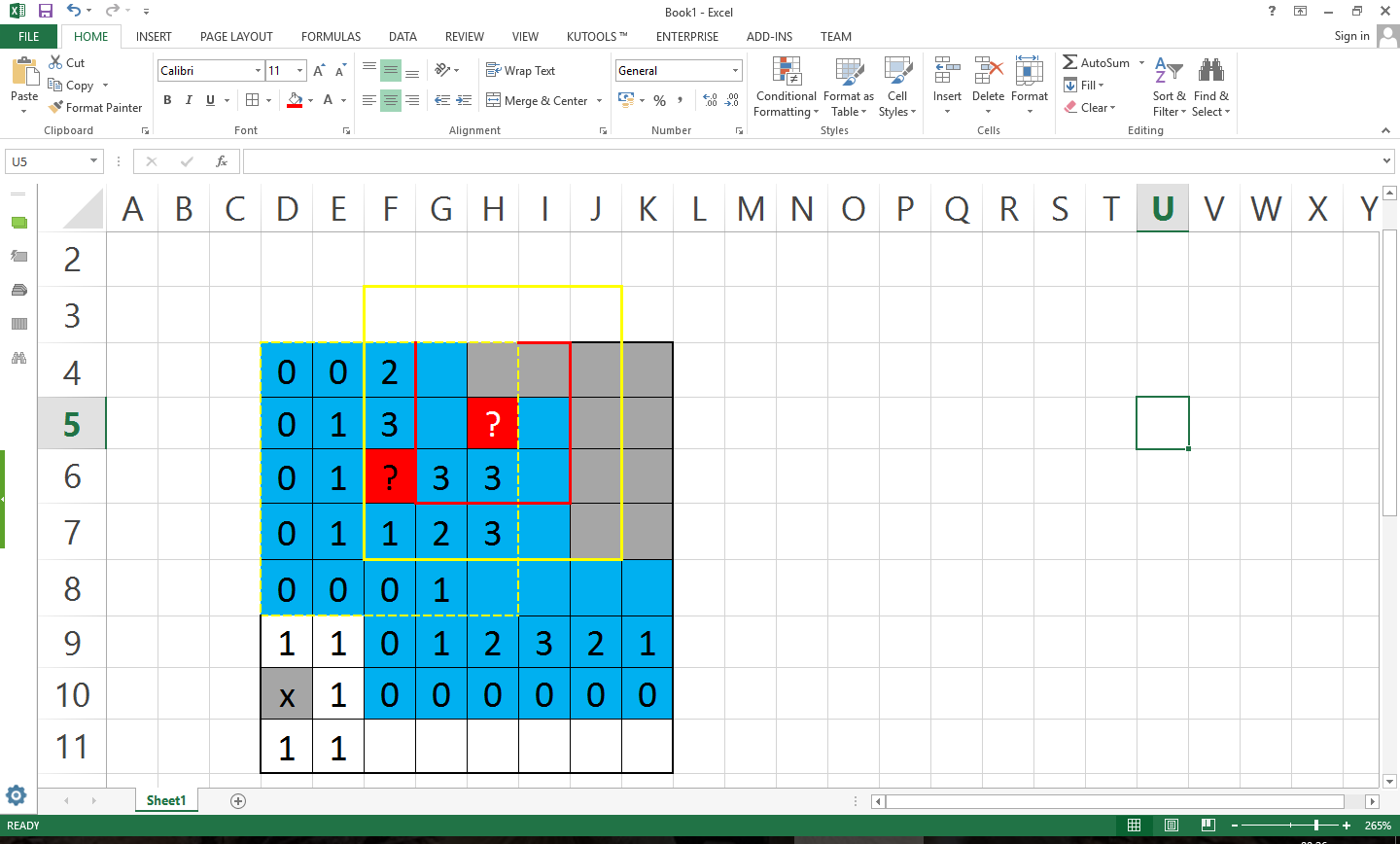
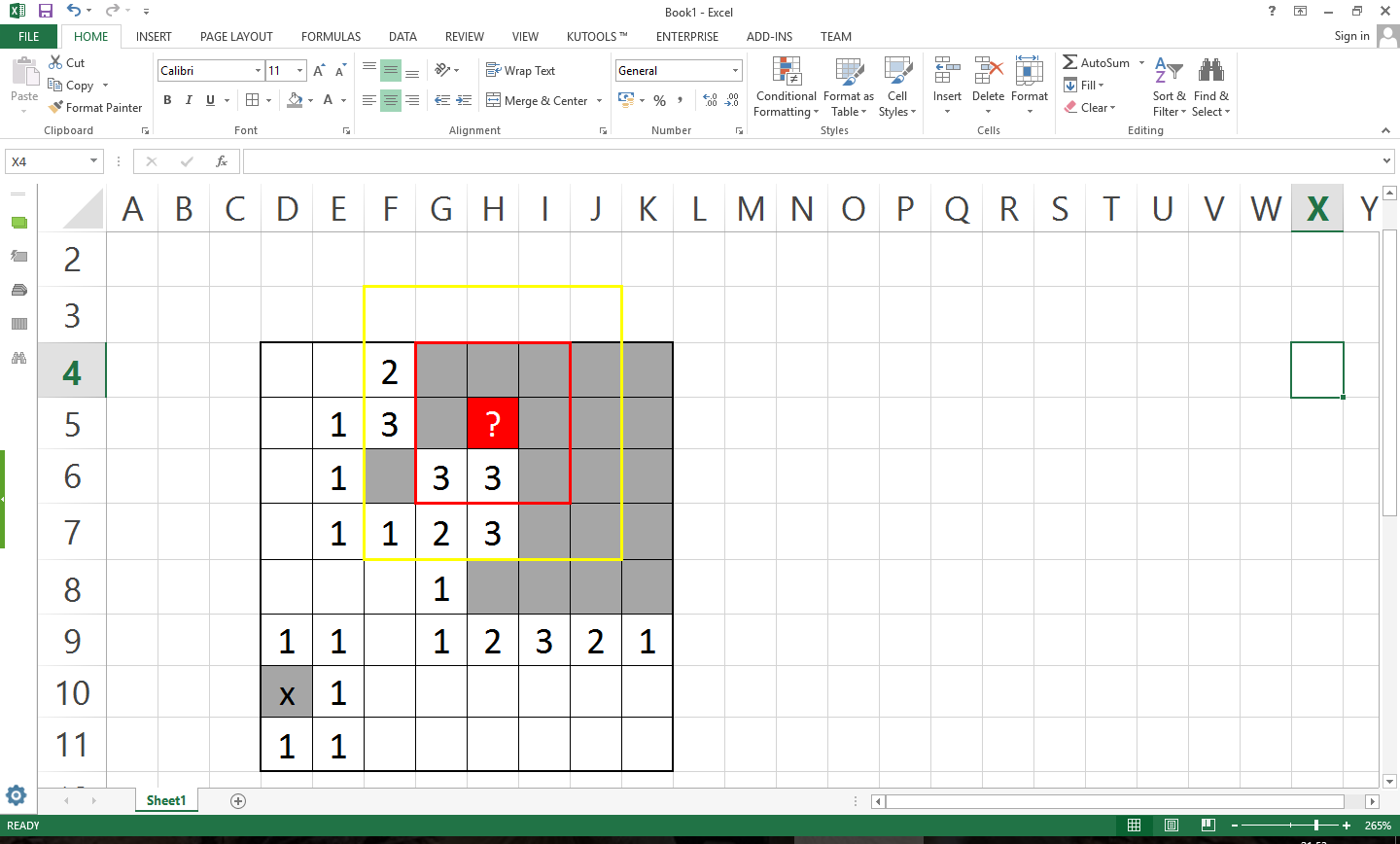
Due to my lack of access to any supercomputers and hyper-speed access memory I had to smartly design how my cases were recorded to optimise computation time without losing a massive amount of information.

Figure 6‑1: A cell of undetermined value (highlighted in red) and two regions of interest with varying diameters

Figure 6‑2: Two different cells of undetermined value, regions of interest and a zone of interest which a normal human might use

Figure 6-1

Figure 6-2

In figure 6-1 we can see a cell in which we are interested in knowing the value of. In the region of influence shown in red, a 3\*3 diameter only shows us a partial picture. We know nothing of the other side of the threes on the bottom row and there is little information we can gather from said information. The other region in figure 6-1 is approximately three times bigger being a 5\*5 and contains a lot more information for us to play with. While this is not a full picture, if the undetermined cell is not a mine, there is a high possibility of seeing a cell which is a mine. It is this gain in information which I harness as the CBRs memories that allow it to develop and learn. Unfortunately this 5\*5 area may not always be enough. While in the case of the black question-marked cell of interest in figure 4-2 a 5\*5 is ample area, the original cell we inspected lacks the detail an AI may require to make sound judgement on it.

Highlighted in blue (on figure 6-2) we see the area in which a human player may observe to find the determined value of the cell in question. This same human player may deduct that there are actually easier cells to determine on the board and abandon this train of thought and solve these easier clues in hopes of shedding more light on the situation after.

As I established a 3\*3 area is too small and for much of the time a 5\*5 will suffice. But then comes the question of how many possible cases are there? Using the 3\*3 case as an example we can assume in an absolute worst case scenario there are 108 possible cases. 10 different symbols {0,1…7,8, unknown} and 8 cells which can be populated with these 10 symbols. This gives us a maximum casebase size of 10 million cases. In table 6-1 and figure 6-3 I have shown how a meagre choice in case size can have a dramatic effect on the database.

Table 6‑6‑1: Possible Case Sizes and their Maximum Case size

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Case Size | 3\*3 | 5\*5 | 7\*7 | 3\*5 |
| Max No. Cases | 108 | 1024 | 1041 | 1014 |

Figure 6‑3: A log. graph visually showing the size of each casebase

The difference between the 3\*3 and 5\*5 casebase sizes is herculean in nature, being 1016, or ten quadrillion cases, which immediately removes the thought of using a 7\*7 case and becomes clear that in order to achieve any kind of reasonable learning time a compromise must occur. Previously I outlined ways which a human player may go about playing Minesweeper and as I am not aware of any alien Minesweeper strategies it only makes sense to learn from what I do know. The blue region of influence in figure 6-2 covers only approximately half of the 5\*5 region shown in the yellow solid border. As I discussed earlier a human player may want to expand out from the knowledge that they already know rather than stabbing in the dark. Because of this, it is very uncommon to have a large amount of data on all sides to work with. It is because of this that the concept of a 3\*5 case comes into view.

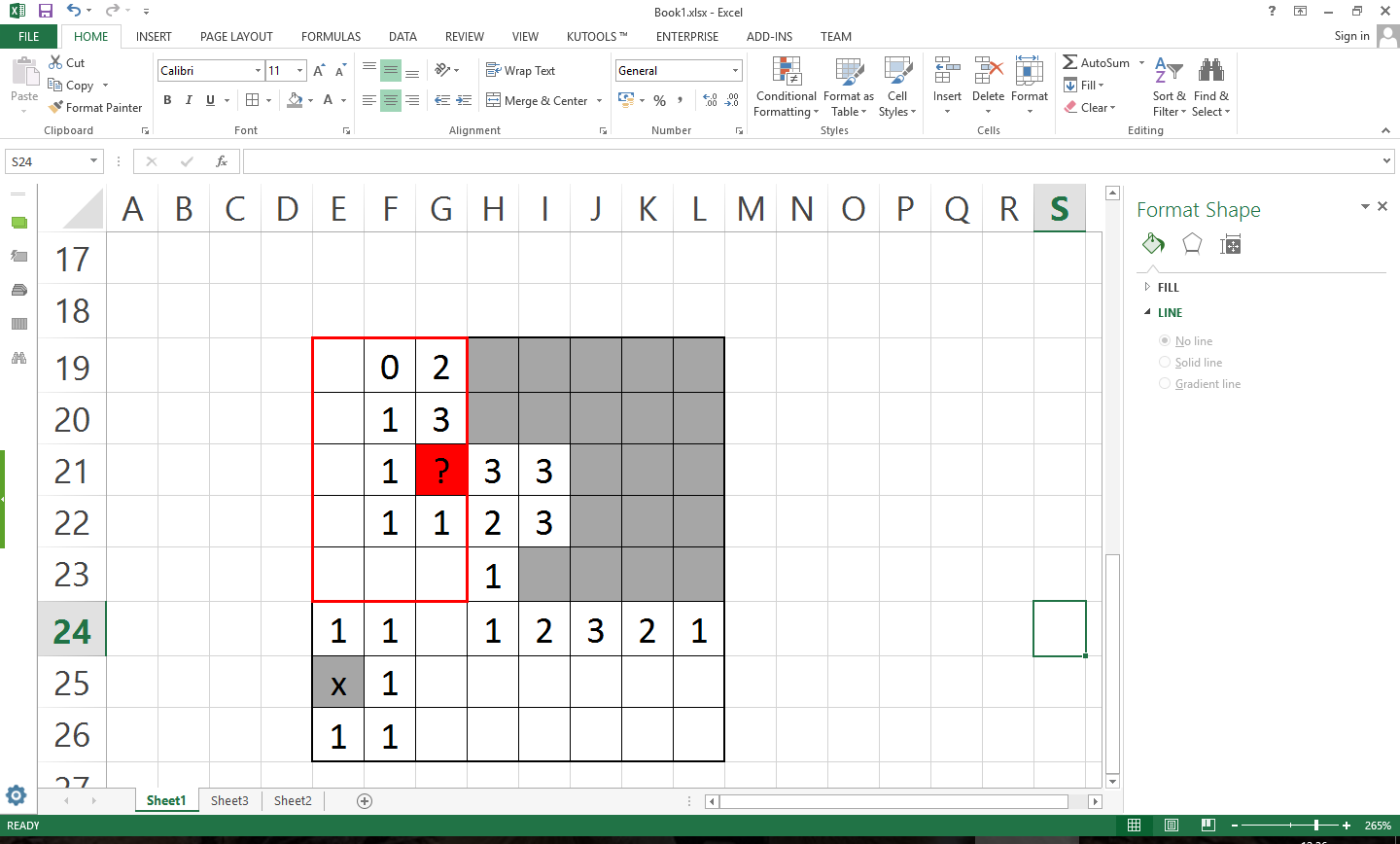
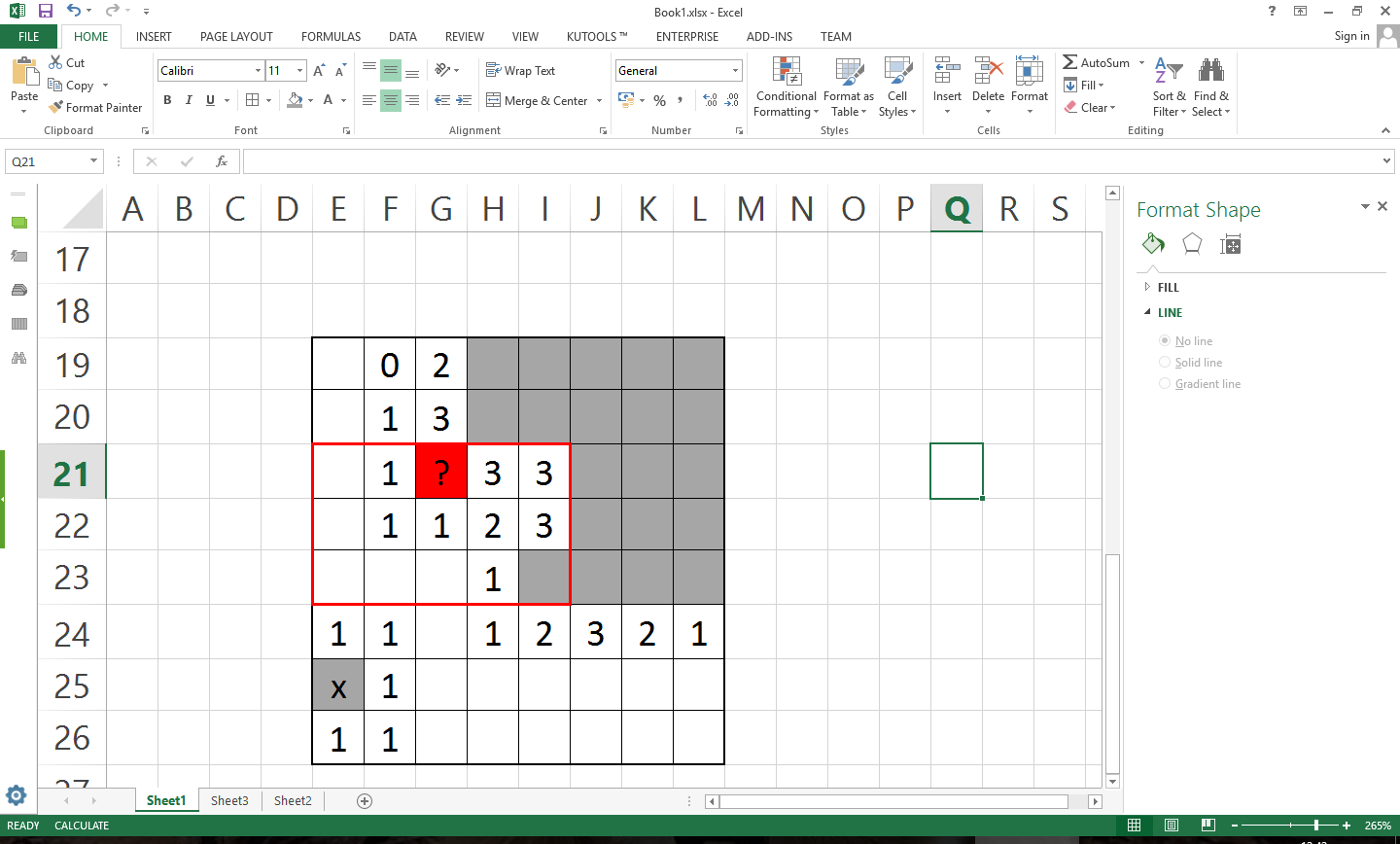
 

Figure 6‑4: A 3\*5 case examples

As seen in figure 6-4 by using a 3\*5 case with the case on a middle edge cell can indeed shed enough information in order to determine if a cell is a mine or not. By using all four rotations over every single unknown cell over the board it is more likely than not that one of these regions examined will match a useful case in the casebase so long as the casebase is large enough. By using this strategy we keep the casebase size maximum size down while also keeping a lot of the precision of a 5\*5 region of interest which while not perfect, can lead us to safe cells a lot of the time with the careful selection a CBR brings.

# Implementation

## Code Outline and Extracts

To streamline the computational workload on my Minesweeper implementation I chose to go without a GUI and opted to work entirely inside the command line and to print out boards in ASCII.

The boards themselves are three dimensional matrixes of 2\*m+4\*n+4 where m and n are board size parameters. The +4 to the board size represents two rings of cells around the play board which are used to handle retrieving border cases, as implemented in figure 7-1 and as seen in figure 7-2.

The first layer consists of 1 or 0 values indicating whether a cell has been revealed or not respectively. The second layer is an integer from -1 to 8 indicating a mine or the number of mines next to it.



Figure 7‑1: Minesweeper board initialisation code snippet

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  | ? |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Figure 7‑2: A 5\*5 sample with a 5\*3 case/region of interest indicated by the dotted line with a cell of interest in the corner of the board indicated by the thick black border, green cells being part of the case, grey cells being part of the case but will always be null values and cyan cells which are part of the board but not part of the case region.

Upon the first cell being revealed a function is called which generates all mines and adjacency clues in spaces which aren’t in the two cell border previously described. A mine cannot be generated in the first revealed cells, as such, the player can never lose on the first turn as seen in figure 7-3.

The user is sequentially requested to input coordinate values of the cells they’d like to reveal. The reveal value is flipped from 0 to 1 and the game continues.

Under the assumption that a -1 is revealed, a piece of code runs which destroys the board instance and runs the lose script. Under the assumption that the remaining cells after one being revealed is equal to the number of mines on the board, the board is solved and the win script is ran.

Under the assumption a 0 cell value is revealed, the zero-flush code is ran, revealing all cells around the said zero cell. To this end if a 0 is next to a 0 then the same zero-flush activates again, flushing all the zeroes in a recursive fashion as seen in the switch statement of figure 7-4.



Figure 7‑3: Mine and clue generation code snippet



Figure 7‑4: Cell reveal and zero-flush code snippet

The case based reasoning AI resolves its problems through recalling similar cases from the past, from its memories. To this end I had to implement an embedded database, for this I chose objectdb, an enterprise level software for persisting objects in a database like fashion rather than using the objects as an intermediate step to translating it to a more traditional database solutions, as other Object-relational mapping softwares do.

The database consists of the case stored, the amount of times the case has been used, the amount of times the case has been successful and the percentage of the time it has been successful, calculated from the previous two parameters.

The case itself is a 2-dimensional array which is a flattened version of the board’s 3-dimensional space via the code seen in figure 7-5 or any of its rotations. These are used down the line to reduce computation time by simplifying the cases.



Figure 7‑5: An example of a compression from a subset of a 3-dimesional to a 2-dimensional array with the cell of interest on the bottom row

The eventual casebase is actually a series of arraylists, as figure 7-6 shows, one to document the last version of the casebase drawn from a permanent database and lists to document changes and additions to the casebase. As will be outlined in the next section, this was to optimise database access time, thus speeding up the CBR’s learning rate overall.



Figure 7‑6: The final Arraylists used as the CBR's casebase

The learner acts for the most part near exactly as a human player playing the game would. Upon being prompted to input coordinates for the next cell to be revealed, the AI cycles from the top left to the bottom right cell. The CBR looks at each cell and comparing its four regions of interest, rotated around the cell of interest, as seen in figure 7-7 and 7-8, to the cases it has stored in its casebase. From here it finds a match to the region of interest which has a win-rate above a certain threshold.

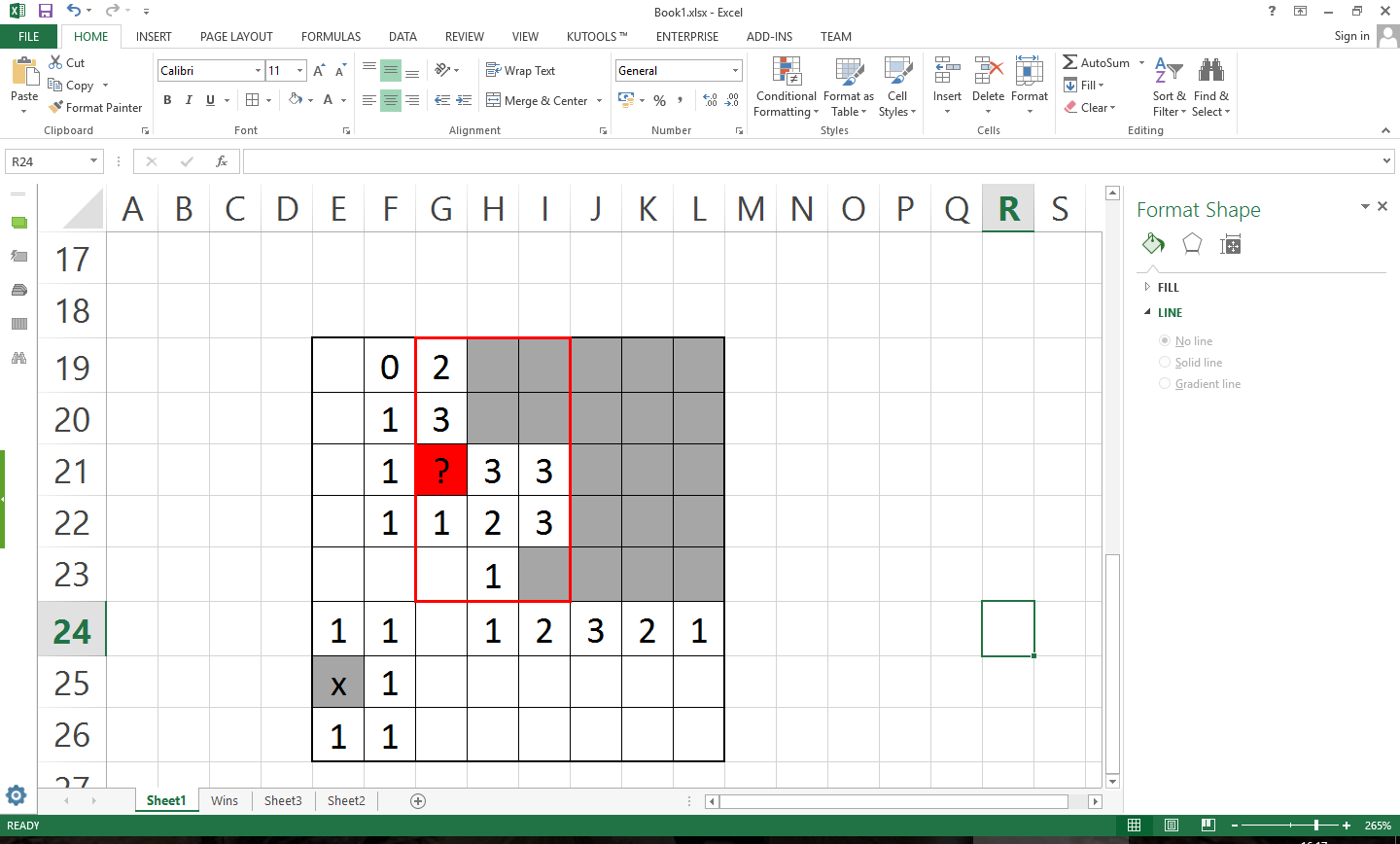
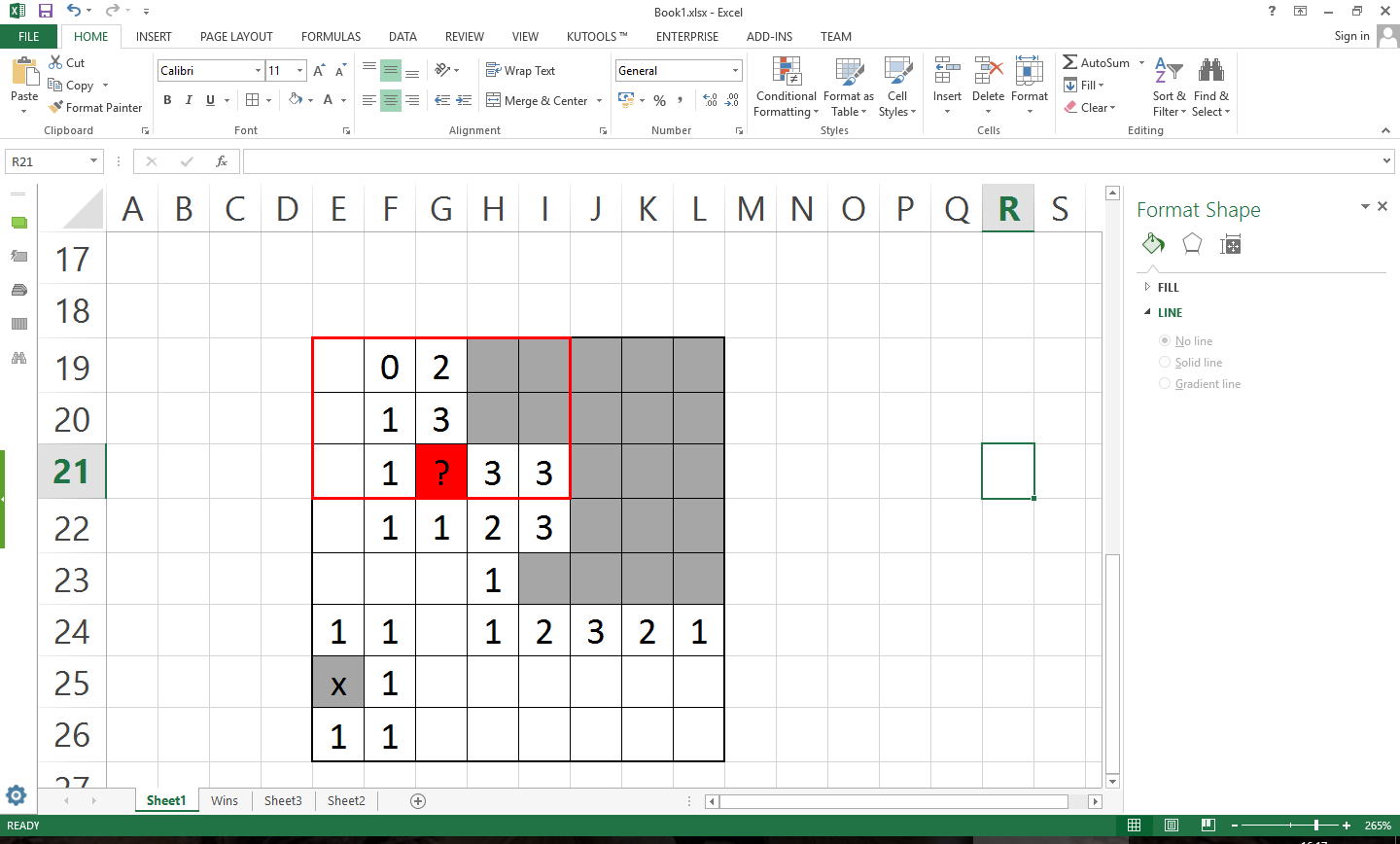
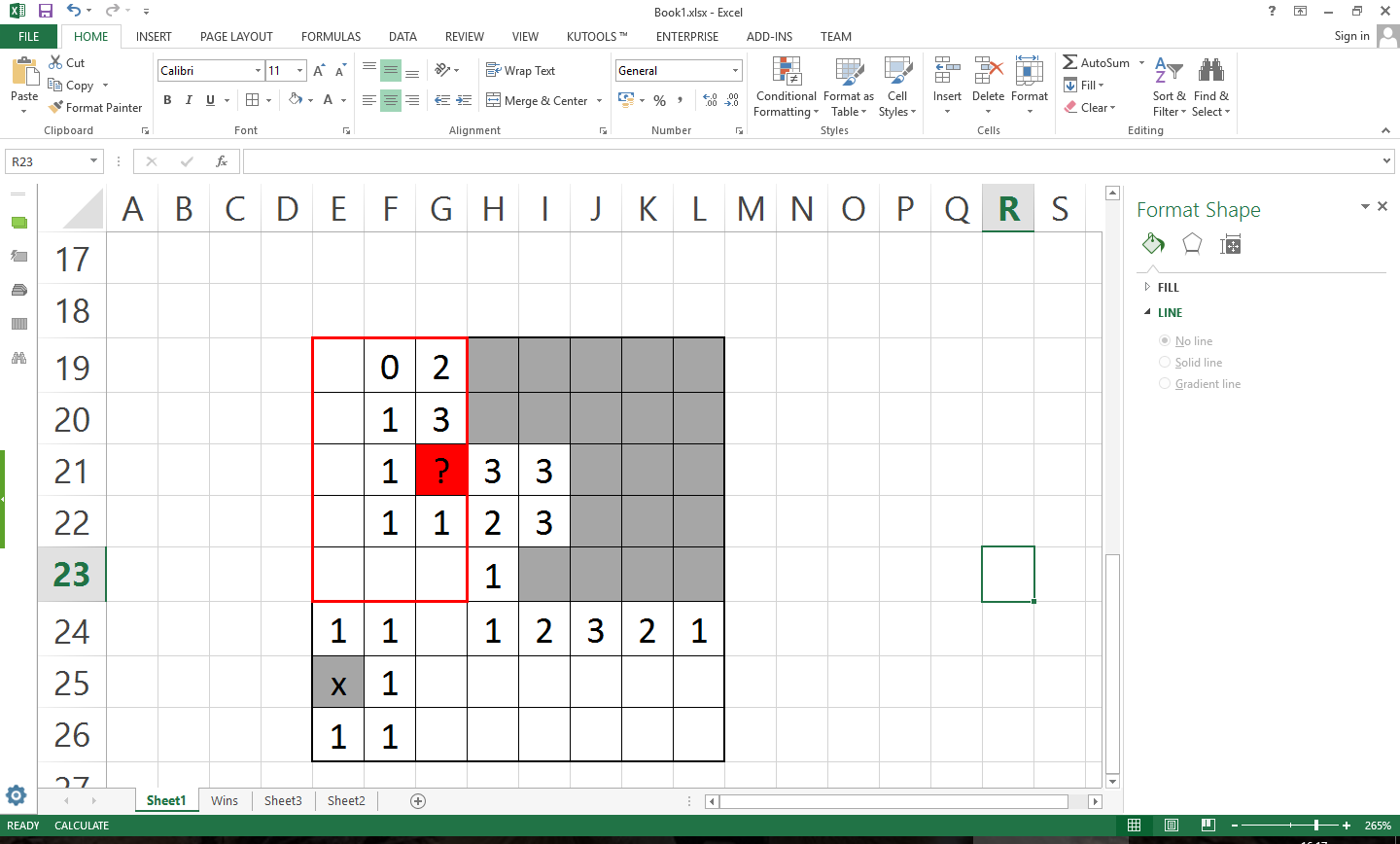
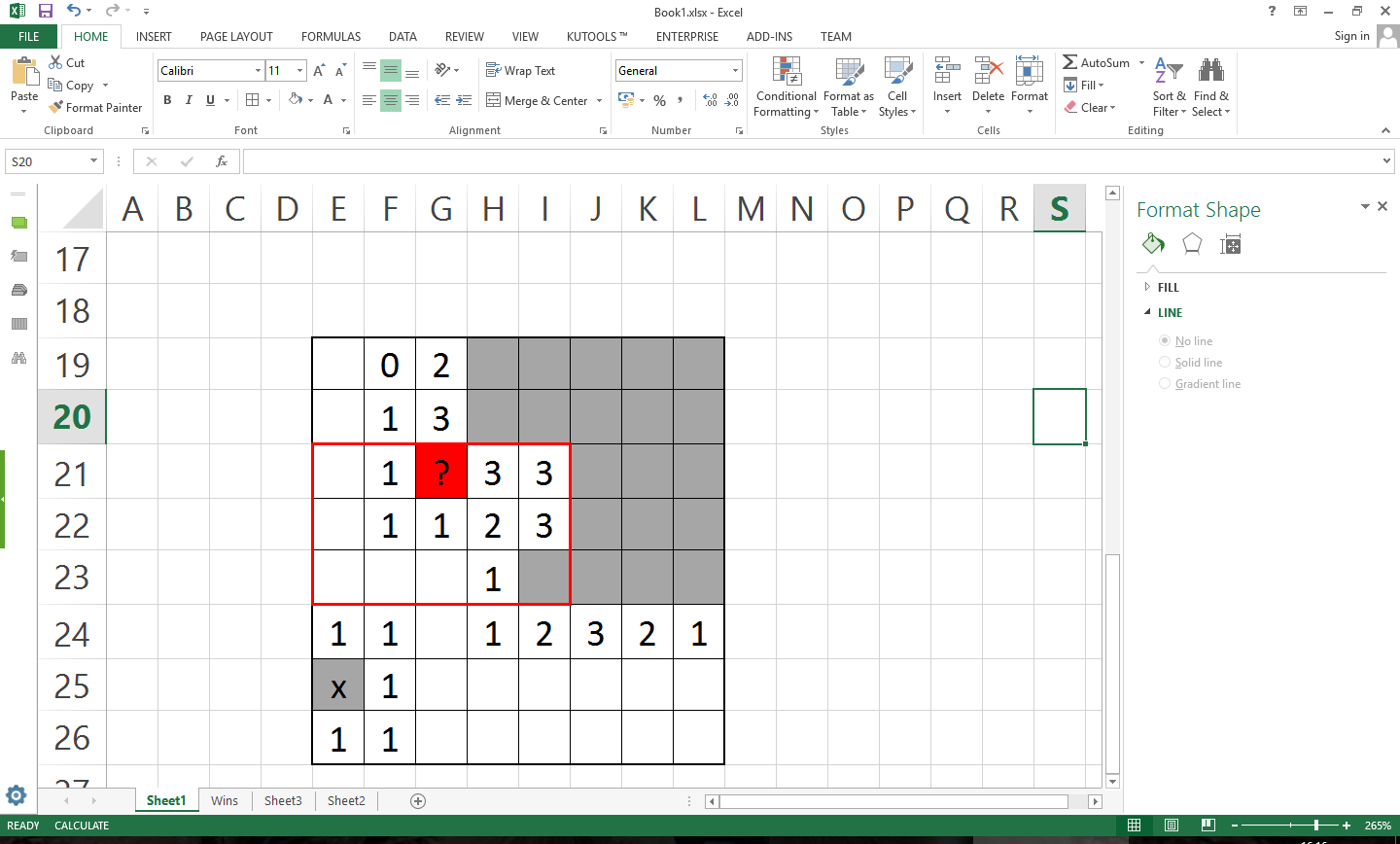


Figure 7‑7: The 4 regions of interest checked by my CBR.

The method that exactly does the comparison is ‘deepEquals’, which recursively compares all single elements to its opposite at the same position ensuring that the two arrays being compared are identical. As mentioned previously I had flattened the cases in the casebase and by doing so cut each individual deepEquals comparison time down by 50%.



Figure 7‑8: A code snippet of the CBR searching the casebase for a match

In addition, in the edge detection version of the Minesweeper Learner I also implemented a score system which also cycled through 19 to 0, the highest to lowest edge scores to find a match. By putting this loop outside of the comparison (as seen in figure 7-9) and having a flag which no longer checks for more comparisons if one is found in a previous edge score loop it means that this version of the CBR prefers a higher edge score than a higher win-rate and will disregard any further cases of lower edge scores once a match has been found.



Figure 7‑9: A code snippet showing the additions made to the CBRs decision process to factor in edge scores

Upon attaining the coordinates, the AI reveals the cell at those coordinates. If the cell is not a mine, then the AI learns from the experience and continues solving the board, if it is a mine then the AI wins or loses, it is given a new board to solve.

After a decision the CBR learns from its experience. There four different outcomes after a decision: the outcome where the learner picks a new case unknown to the casebase in which case it learns from the experience, the outcome where the learner picks a case that it has learnt and does not step on a mine, it’s opposite where it picks a case where it has learnt but does step on a mine and the opposite of the first situation, a situation where it does not find a match, reveals a random cell and exposes a mine, thus terminating that game.

In the first situation where it does not find a match but does randomly select a safe cell then the code in figure 7-10 is ran.

From here it checks the four regions of interest (as outlined in figure 7-7) for duplicates, stores the non-duplicate cases as new cases and increments the appropriate statistics of any duplicate in either of the two arraylists used by the CBR. This ‘redemption code’ was included so that if a case does drop below the threshold it means that they are not expelled from the match selection process forever. If a case is, for a lack of a better word, ‘unlucky’ and drops below the win-rate threshold early in its existence inside the casebase it may be cast aside but could still actually be of use. While this is a fringe case, by having this get-out clause only during random selections it allows truly useful cases to eventually return to above the selection threshold.



Figure 7‑10: A code snippet that shows how the CBR adding new cases to the casebase

Under the assumption a case is found and successful reveals a non-mine cell then the appropriate cases win-rates and usage statistics are adjusted accordingly, as seen in figure 7-11.

The redemption code was purposefully excluded here as to remove the possibility of producing false positives, which appear far more often than in the cases redeemed in figure 7-10.



Figure 7‑11: A code snippet showing what happens after a case is found and reveals a safe cell

If a case were to lead the CBR astray and reveals a mine then there are two variations of code that handle this event. The first, figure 7-12, is the implementation I used in the forgetful CBR as it checks where this case resides and subsequently removes it from there, thus forgetting the case it it’s entirety.



Figure 7‑12: The code which the CBR uses to 'forget' a case

In the second implementation, figure 7-13 the uses and chances of that case are incremented but not successes, yielding a lower chance.



Figure 7‑13: The code which increments the usage statistics after a mine is revealed

Once every 1000 games a ‘save’ happens. During this time any changes made to the casebase and any new cases that have been found are persisted to the main database. This happens in three steps as seen in figure 7-14. In the first step, any newly discovered cases are added to the database. Secondly, any cases marked for removal are removed (which only happens in the forgetful CBR) and third any changes to case statistics are committed, which happens in the two versions which do not forget. All arraylists are then whipped clean and a fresh casebase is fetched.

It should be noted here that eight different statistics are recorded over these 1000 games and are used for analysis. They are:

* Date and Time
* Games Won this 1000 games
* Database Size
* Latest Cases ID
* Games where a zero-flush was the first move this 1000 games
* Games won where a zero-flush was the first move this 1000 games
* Turns played this 1000 games
* Matching cases found this 1000 games

After the database changes are in place the statistics file is updated in a similar fashion and then all runtime statistics are reset ready for the next 1000 games as well.



Figure 7‑14: A code snippet of the save process

Each CBR implementation was also implemented with two minor adaptations: ‘win-learning’ and ‘out of bounds tokens’.

As in its namesake win-learning is the process in which the CBRs learnt new cases when they successfully revealed a non-mine cell, via the code in figure 7-10. After revealing the cell, if any region in the cell of interests four regions of interest was not already a case, those regions of interest would be added to the arraylist of cases that were to be added to the casebase. This had the effect of speeding up the amount of cases there were in the casebase early on and adding potentially useful yet obscure cases to the casebase later on.

The second minor adaptation was to add another symbol to the cases once flattened, an out-of bounds symbol. Due to my implementation and region of interest shape it meant that there were times where the regions of interest would extend over the edge of the board. In the cases, these would show up to be 9s; unrevealed. Which while technically true in the implementation, are not true in a real game of Minesweeper, as they do not exist at all. This state of non-existance actually gives us information about cells on the edge of the board as it means that there are less cells to which a mine could be next to them.

## Implementation Problems

### Database Integration & Limitations

I had major problems with implementing a storage solution for my CBR as I found while there were many solutions they were either obtuse, strikingly slow or were not portable with ease.

Javadb and apache were excellent candidates to use but due to their lack in easy portability they were both eventually disregarded. It was here that I discovered Java Persistence API (JPA) and MySQL, but as mentioned earlier, after a long struggle I eventually found and chose objectDB with JPA due to its ease of use and also its speed (Jpab.org, 2016).

### Database Optimisation

As I mentioned in section 5, my original implementation of a CBR had a glaring issue with it. After a long while and using the profiling tool ‘Visual VM’ as seen in figure 7-15 and 7-16, I was able to determine that a large portion of my CBRs computation time was being wasted on simply the construction of database handlers and get requests, which spurred me to create the ‘save’ method. By having only one Entity manager per learner and by using the save function, these functions go from being called every move of every game to once in a thousand games, speeding up my CBR immensely.

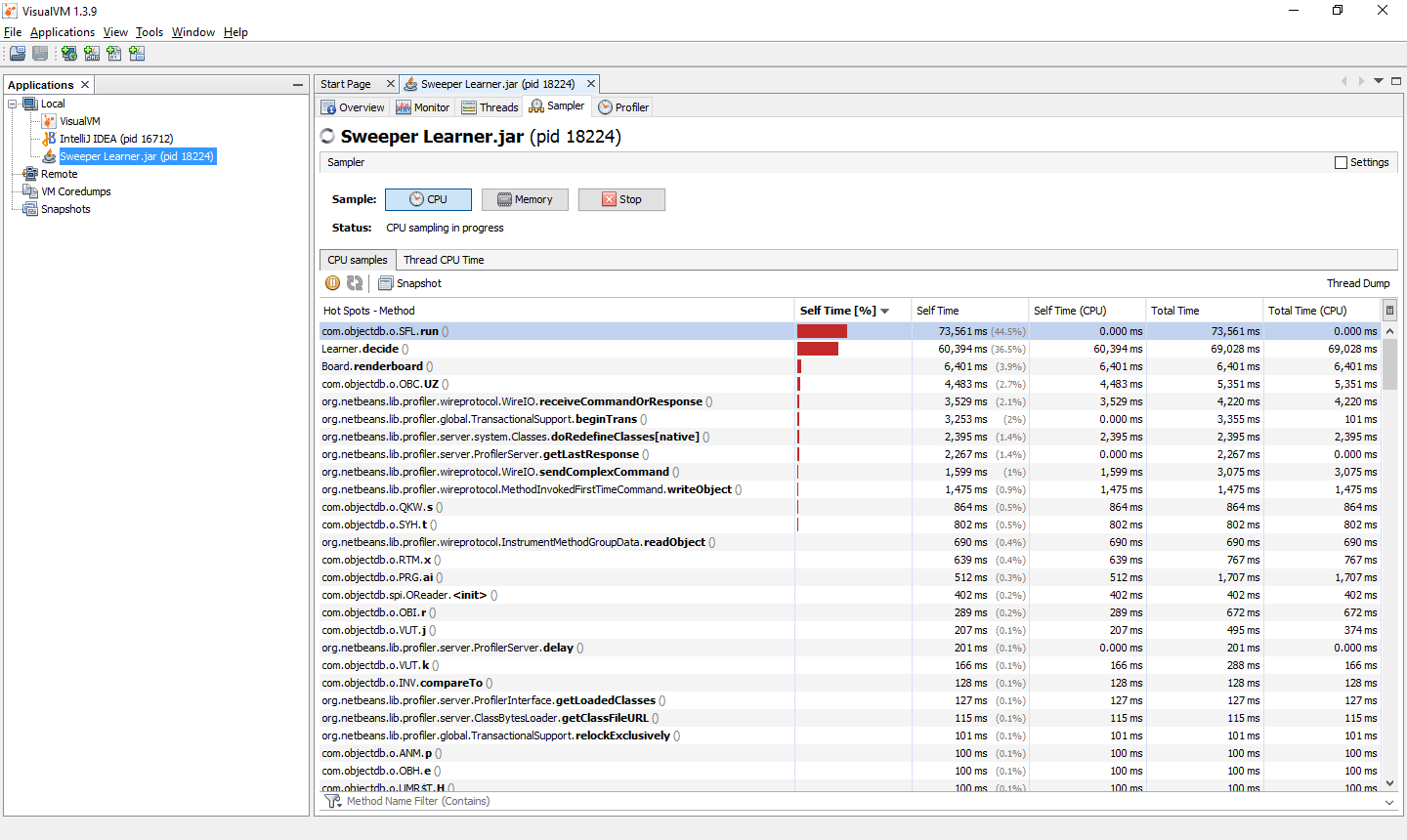


Figure 7‑15: Profiler showing how objectdb functions require the majority of my computation time

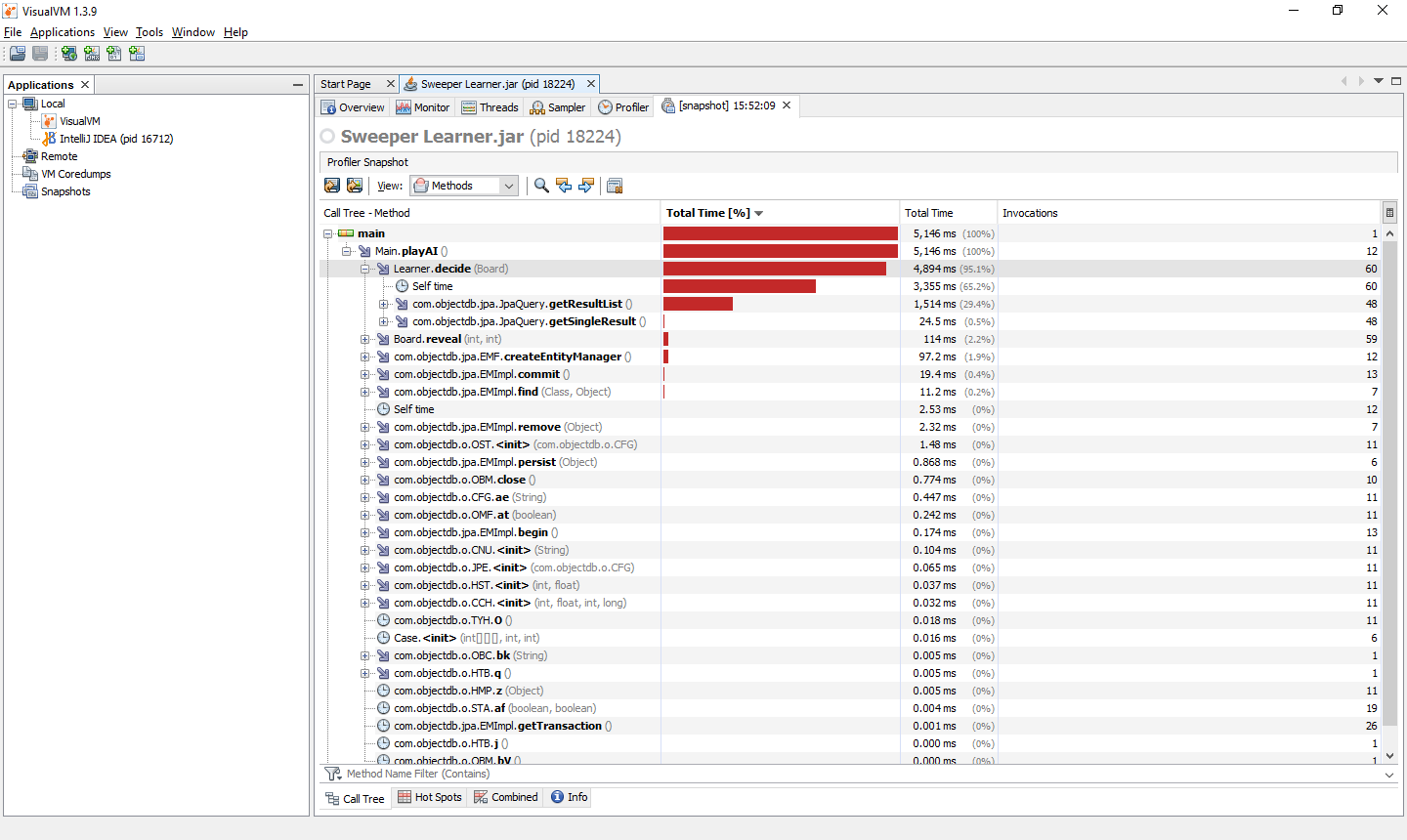


Figure 7‑16: A snapshot of the playAI method computation breakdown

# Testing

As a preface, all testing was done on machines with SSD storage and with an intel i5 processor or of equivalent strength with high CPU priority.

## Minor Adaptation Testing

As mentioned in my requirements, by generating the Minesweeper board when the cell is revealed, this meant that the first move was always successful. Because of this it meant that my forgetful learner always learnt a fully blank case at the start of the game if that case was not already present in the casebase effectively introduced a huge hole for false positives to flood in. Preliminary results showed that the CBR never failed to waste an opportunity to learn something useful by learning a bad but common case with no information, the first move of every game. This trend continued and after an eight hour test, the casebase grew to a size of 2067 actual cases but with a total of 5,255,779 learnt cases, 0.0004% case retention, which promptly led me to patch this by bug by only allowing cases to be learnt from the second move onwards. While this still lead to false positives, it decreased the amount dramatically.

In the other two CBR implementations, this caused the casebase to grow more rapidly and to capture new cases, just as it was designed to. This is due to any harmful cases produced was soon filtered out by being selected, losing and thusly dipping below the required win-rate threshold.

The second adaptation I outlined in the previous chapter was largely unsuccessful not due to the theory itself but due to physical limitations. By introducing an ‘out-of-bounds’ (OOB) token I introduced two different problems. Firstly it increased the absolute size of my casebase from 1014 to 1114, effectively quadrupling its size. Secondly it created an effective fracture in my actual casebase. Around the edge of my game boards several cases on each cell overlap the edge, ensuring that there is indeed OOB tokens in the region of interest. This means that only cases with OOB tokens should be checked. In figure 8-1 yellow cells have three regions of interest which contain OOB tokens and all regions of interest in the red cells contain an OOB token. This means that of the 256 regions of interest checked on a beginner Minesweeper board, 160 of them have out of bounds tokens and 96 do not. This effectively meant that half of my deciding cycle would be wasted as there was no chance of these matching and meant that there was more information that my CBR had to learn in the same amount of time. If I had more time I would of implemented two separate databases for these to save time during the comparison phase of my CBR.

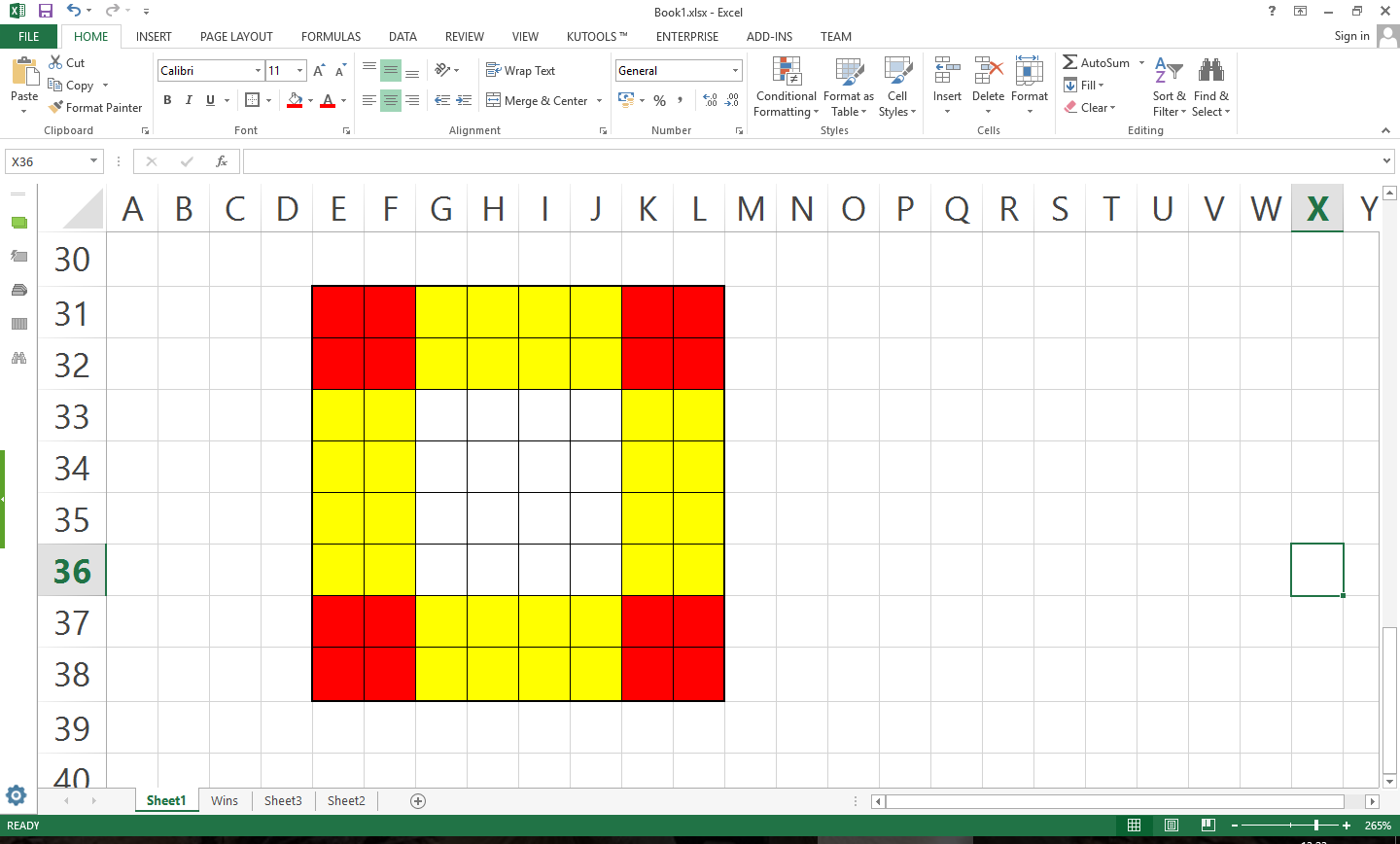


Figure 8‑1: A beginner Minesweeper board with the shaded cells being areas where there would be an out of bounds token

## Testing Theory

In my Non-forgetful CBRs I tested three different win-rate thresholds on a ‘classic’ beginner board (8 by 8 with 10 mines) to compare which one would perform best and to use in further comparisons. It should be noted that all three used win-learning.

The first threshold I tested essentially allowed any new case two consecutive failed uses after its initial use before ruling it out and all other cases have to have chances about as good as a coin-flip to be considered for selection. This was the mode I originally tested and built my casebase up from.

This methodology was developed with the idea that a 50/50 chance was too relaxing a methodology, while it may serve well to generate a good casebase, the question becomes: can we do better? Can we get a better winrate by adjusting this threshold? Thus I decided on two other threshold values to test my non-forgetful CBRs on. 66% and 100% thresholds.

At 66% it means that any case that is considered needs to at least mostly be alright to use and any new case that dips below the threshold does not have far to climb back up in order to be considered again.

At 100% threshold it means that only cases that win will be considered and any case that loses is locked out of the selection process forever. The reasoning behind this methodology is similar to that of the forgetful CBR, what use is a case if it leads the sweeper to lose?

## Analysis of results

### Forgetful CBR

As previously mentioned, my forgetful CBRs minor adaptations proved to be rather fruitless, and much to my dismay, a tragic realisation came to light.

Figure 8-2 shows my Forgetful CBR’s progress after about 100 hours of learning. It becomes apparent that even if there is a ‘critical mass’, it will take a long time for my CBR in its current state to reach it.

Figure 8‑2: A graph showing how the number of wins my forgetful CBR achieves over the amount of games played

It should be noted however, this methodology is not without merit, in figure 8-3 we can see that the case retention has flattened out to about 20% while in figure 8-4 we can see the casebase size is still growing healthily, meaning that ideally speaking in the future, once the casebase has stopped growing and the case retention rate has stabilised, my CBR will have reached it’s critical mass. Though its final win-rate may not be as fantastic as I first imagined.

Figure 8‑3: A graph showing how much of the casebase my forgetful CBR actually retained

Figure 8‑4: A graph showing my forgetful CBRs casebase size over the amount of games played

Predictably, the use of an OOB token stifled the learning rates dramatically, figure 8-5 shows that very little was learnt in the 1.5 million games of Minesweeper played. Surprisingly, Figure 8-6 shows that even without OOB tokens in the cases, the win-learning adaptation did not yield good results either.

Figure 8‑5: A graph showing how the number of wins my forgetful CBR with both adaptations achieves over the amount of games played

Figure 8‑6: A graph showing how the number of wins my forgetful CBR with the win-learning adaptation achieves over the amount of games played

The overall lower win-rate in regards to win-learning can be attributed to many bad cases being repeatedly added to the casebase through the win-learning process. In figure 8-7 we can see that despite being ran for about half as many games, the win-learning database has a comparable amount of cases, which while many of them being good, a lot of them may be obscure cases which have only fringe uses.

The kink in the otherwise smooth curved line on figure 8-7 is due to a windows update force-restarting program which shed light on a bug related to the start-up procedure on that specific version of the CBR, which while never remedied was never encountered again as the CBR was never started up again after it was closed.

Figure 8‑7: A graph that shows the size of the casebase as the amount of games my forgetful CBR plays

### Traditional CBR

While my forgetful CBR showed promise but little result, my other CBRs show both promise and result. Figure 8-8 shows us in the same amount of time, approximately 100 hours, my traditional CBR approach was able to achieve win-rates 10 times higher than its forgetful counterpart in under a tenth of the games and was still growing at a respectable rate. It should be noted that this version of the CBR had to cycle through all cases in the casebase on all cells in all rotations, up to (x\*y)4n comparisonswhere x and y are the board dimensions and n is the number of cases in the casebase. While it only needs to use the deepequals method on cases above the win-threshold, it still means that the exponential time complexity of this CBR methodology makes casebase growth literally exponentially slower.

Figure 8‑8: A graph showing how the number of wins my traditional CBR with the win-learning adaptation achieves over the amount of games played

As of writing this report, my traditional CBR shows no sign of its win-rate flat-lining (at 400 hours of learning time), it continues to go up and up but as just mentioned, at a glacial pace. With a casebase of just over 250,000 recorded cases, this version of my CBR takes just shy of 7 hours to complete a thousand games, or about 2 games per minute.

In figure 8-9 I have highlighted a particular region of interest, a breakthrough case or series of cases are discovered, which give my CBR an extra tool to work with, boosting its win-rate an amount ahead of the average predicted win-rate. While there have been these kinds of spikes before, this localised group are almost all above the orange trend line for about 10 thousand games which seems to be too large a selection to be coincidentally easy boards. Due to this version of my CBRs win-rate not flat-lining yet I can only conclude that my CBR underwent an unusual amount of growth in a short period of time from board patterns it had not seen up to that point.

Figure 8‑9: A graph showing how the number of wins my traditional CBR with the win-learning adaptation achieves over the amount of games played with a highlighted area of interesting growth

### Edge Detection CBR

My third methodology explored gave even better results. While not exclusively a CBR approach anymore, the introduction of this scoring system accidentally gave rise to a hierarchical structure which I first thought increased its win-rate but in fact did something I did not predict instead.

With all of these thousand game cycles taking less than an hour each; after the same amount of learning time as my ‘vanilla’ CBR, my edge detection CBR was able to play over four times as many games and as such has had more games or ‘effective time’ to learn, having the ability to win over 250 games in every thousand as seen near the end of figure 8-10.

Figure 8‑10: A graph showing how the number of wins my edge-detection CBR with the win-learning adaptation achieves over the amount of games played

In figure 8-11 we see that the CBR has essentially had the same amount of non-random clicks for the last half a million games but in figure 9-12 we see that the number of cases, while slowing down slightly is still expanding at a respectable rate. I realised that this means that while the casebase is still growing, these are becoming more and more niche cases as time goes on, as it finds patterns that it has never seen before. This also means that due to these cases becoming more and more niche it means that it will become harder and harder for my CBR to improve its win-rate, though not impossible to have larger jumps as discussed previously.

Figure 8‑11: A graph that shows the percentage of Non-random clicks made per 1000 games

Figure 8‑12: A graph that shows how my Edge-Detection CBRs casebase grows as the CBR played

Though this all begs the question, how do the win-rates compare after the same amount of games have passed? In figure 8-13 we see my edge-detection CBRs data after 146 thousand games, the same amount as my traditional CBR has currently played. We can see several features which both figure 8-13 and 8-9 share. A steady increase in wins, lack of any flat-lining and a sub 16% win-rate. For clarity I have included a graph of both trend lines. Both trend lines are modelled on a 6th power polynomial and as we can see from figure 8-14, they are nearly identical.

Figure 8‑13: A graph showing how the number of wins my edge-detection CBR with the win-learning adaptation achieves over the amount of games played for its first 146 thousand games

Figure 8‑14: A graph which shows the trend line of both my Traditional and Edge-Detection CBRs modelled as 6th power polynomials

From this, there is no evidence that my edge-detection adaptation is any better at winning at Minesweeper but it is a whole lot faster. This does not say about the future of my CBRs though, while in figure 8-10 it is clear that the edge-detection learning has slowed down dramatically neither of the two CBRs have essentially reached their soft-maximum win-rate. For all anyone knows my traditional CBR could, given enough time, continue to grow and learn or even have a higher win-rate than its adapted counterpart.

The hierarchical structure previously mentioned leads me to suspect that as my CBRs matures and grow, the innately good cases float to the top and the bad ones eventually sink to the bottom of this list. Intuitively, these good cases carry the most useful information with them and thus usually have a higher ‘edge-score’. This leads me to theorise that the edge-detection adaptation of the CBR is not just a better selector which results in a better win-rate but is actually also more of a sorting method that allows the casebase to find these good cases quicker.

As my best learner, it was this CBR that I chose to explore different win-rate thresholds on. Like I mentioned a little while ago the two other win-rate selection thresholds are 66% and 100%. Both of these CBRs were ran on their own separate casebases that were copies of the original edge detection CBR casebase after 300 hours of learning.

In the 100 hours of testing these CBRs the 66% one played 68 thousand games and the 100% threshold one played 20 thousand games. As opposed to the 113 thousand played by the original in the same amount of time. We can assume this is because as the selection threshold increases, my CBRs become more and more strict, resulting in them having to dig through the casebase more thoroughly to find an match. Unsurprisingly, just as with my traditional CBR and my edge-detection CBR, there was no substantial difference between the original and the 66% threshold win-rates and any difference is due to the amount of games played during that time. The 100% threshold CBR however did result in very poor results of sub 15%, showing a severe lack of 100% effective cases in the casebase and an ample supply of false positives.

Figure 8‑15: My original edge-detection CBR (with win-learning) and its results for the 100 hours proceeding

Figure 8‑16: My edge-detection CBR (with win-learning) with a 66% selection threshold

There is however a silver lining, in table 8-1 we see that the 66% threshold CBR on average played 400 more moves every thousand games, making an extra move roughly once every other game! This shows us that there are a group of cases in the casebase with a chance of hitting a mine above 49% but below 66% and that by increasing the selection threshold it stops these potentially hazardous cases from being used. This has the potentially effect that under the assumption the CBRs had more training time, the 66% threshold version could have higher win-rates than its 49% counterpart.

Table 8‑1: A table showing the average amount of moves per thousand games over the 100 hour period of learning

|  |  |  |
| --- | --- | --- |
| Edge-Detection CBR | 66% Threshold | 100% Threshold |
| 12938 | 13317 | 4836 |

## Other Comparisons

In order to measure my learner’s effectiveness we have to compare it to other implementations, both learners and solvers and also human players to find out how good it really is. For these purposes I compared my best CBR, the edge detection CBR on a classical beginner board.

Due to the fact that most human players compare speeds rather than win-rates I could not find an accurate human win-rate. To this end I had to conduct a small scale informal survey. In this survey of 10 people, which played 10 games each, three of them (including myself) said that they had played or did play Minesweeper on a regular basis while the other seven only knew the rules but not very familiar with the game. To this end I split my volunteers into two groups: adepts and amateurs. The adept group achieved an average score of 66.66% and the amateur group achieved a win-rate of 30%.

Figure 8‑17: A bar chart showing win-rates of the player groups, solver and learners

In figure 8-17 we can see that my best CBR has grown to the level of nearly as good as an amateur human player but still has a long way to go until it could be competitive with the multi-relational learning method or a preprogramed solver.

I also did short tests on intermediate (a 16 by 16 board with 40 mines) and expert (a 16\*30 board with 99 mines) minesweeper difficulties. Over the course of 12 hours each CBR managed to play approximately 2 thousand games with the intermediate one winning 2 games and the expert one winning none. From inspection of the winning boards it is easy to see that certain board configurations are more likely to occur than on beginner boards, resulting in substantially lower win-rates. Similarly with a higher mine-density in expert boards, it is very likely that the CBR ran into many new cases and board states that it had not experienced as of yet. I also believe that very few boards were even played was a result of a larger game board, increasing the number of comparisons needed exponentially.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | \_ | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |  | 0 | 0 | 1 | 1 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | \_ | 3 |
| 2 | 2 | 2 | 1 | 1 | 1 | 1 | \_ | 3 | 3 | 2 | 1 | 0 | 1 | \_ | 1 |  | 0 | 0 | 1 | \_ | 2 | \_ | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | \_ | \_ |
| 1 | \_ | 1 | 1 | \_ | 1 | 2 | 4 | \_ | \_ | \_ | 2 | 1 | 2 | 1 | 1 |  | 0 | 1 | 3 | 3 | 3 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 0 | 1 | 3 | \_ |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | \_ | \_ | 4 | 2 | 2 | \_ | 1 | 0 | 0 |  | 0 | 1 | \_ | \_ | 2 | 0 | 0 | 1 | \_ | 3 | \_ | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 1 | 0 | 1 | 1 | 2 | 1 | 1 |  | 1 | 2 | 4 | \_ | 2 | 0 | 1 | 2 | 3 | \_ | 2 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | \_ | 1 |  | 1 | \_ | 3 | 2 | 2 | 0 | 1 | \_ | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| \_ | 1 | 0 | 2 | \_ | 2 | 0 | 1 | 3 | \_ | 2 | 0 | 0 | 1 | 1 | 1 |  | 1 | 1 | 2 | \_ | 3 | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 1 | 1 | 0 | 0 |
| 3 | 4 | 2 | 3 | \_ | 2 | 0 | 1 | \_ | \_ | 2 | 0 | 0 | 1 | 1 | 1 |  | 1 | 1 | 2 | 2 | \_ | \_ | 1 | 0 | 2 | \_ | \_ | 2 | \_ | 2 | 2 | 2 |
| \_ | \_ | \_ | 3 | 2 | 1 | 0 | 1 | 2 | 2 | 1 | 0 | 0 | 1 | \_ | 1 |  | 1 | \_ | 1 | 1 | 2 | 2 | 1 | 1 | 4 | \_ | 5 | 3 | 1 | 2 | \_ | \_ |
| 2 | 3 | 3 | \_ | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 2 | 2 |  | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 2 | \_ | \_ | \_ | 1 | 0 | 1 | 2 | 2 |
| 1 | 1 | 2 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | \_ | 2 | 1 | 0 | 1 | \_ |  | 1 | 1 | 1 | 1 | 2 | 4 | \_ | 3 | 2 | 3 | 3 | 2 | 1 | 0 | 0 | 0 |
| 2 | \_ | 2 | 0 | 0 | 0 | 1 | \_ | 1 | 2 | 4 | \_ | 3 | 1 | 1 | 1 |  | 1 | \_ | 1 | 1 | \_ | \_ | \_ | 2 | 0 | 0 | 1 | \_ | 1 | 0 | 0 | 0 |
| 2 | \_ | 2 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | \_ | \_ | \_ | 1 | 0 | 0 |  | 1 | 1 | 1 | 1 | 2 | 3 | 2 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 2 | 2 | 1 | 0 | 0 | 1 | 3 | \_ | 2 | 2 | \_ | 5 | 4 | 3 | 1 | 0 |  | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 2 | 1 | 0 | 0 | 0 |
| \_ | 1 | 0 | 0 | 0 | 1 | \_ | \_ | 3 | 1 | 1 | 2 | \_ | \_ | 1 | 0 |  | \_ | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 2 | \_ | \_ | \_ | 2 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 3 | \_ | 2 | 0 | 0 | 1 | 2 | 2 | 1 | 0 |  | 1 | 1 | 0 | 0 | 1 | \_ | 1 | 0 | 2 | \_ | 5 | \_ | 2 | 0 | 0 | 0 |

Figure 8‑18: The two Intermediate games my Edge-Detection CBR with win-learning won

# Specific Case Analyses

Upon examining the edge-detection casebase I found some very interesting discoveries. By far the most used case was what you can see in figure 9-1 with a total of 228,138 tries. This on its own is unremarkable. With a 51% chance (and 2% standard deviation) of the CBRs first click being a zero flush and the (somewhat) low mine density of 15% on a classic beginner board it is only reasonable to see it being used so much. What is remarkable though is that of the 228,138 times this case was used, 228,137 of these clicks were not a mine. Only one of a quarter of a million clicks was a mine.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  | 1 |  |  |
|  |  | ? |  |  |

Figure 9‑1: The most used case in my edge detection CBR

Under the assumption that each of the cells around the revealed cell had an equal chance of being the mine (which the law of averages dictates), it means that the chances of this happening were 5\*10-206029 or in more simple terms, low. Of course this does not account for the edges of the board, which this version of the CBR does not show. Even so, the probability of this happening is astronomically small and led me to do an investigation into my code and other cases. This however showed no sign of error within my code and other cases appeared to be mostly normal. For example in the third most used case, figure 9-2, was used a total of 72,472 times and resulted in a mine not being pressed only 62,299 times, about 86% of the time. This leads me to believe that as strange as it is, this might just be a coincidence… That and my CBR is great at Russian roulette.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 |  |  |
|  |  |  |  |  |
|  |  | ? |  |  |

Figure 9‑2: The third most used case in my edge detection CBR

Of the top 50 most used cases, only two were saved after the first 10 thousand games played and none were saved after the first 100 thousand games.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 0 | 0 | 0 |  |  | 0 | 0 | 0 | 0 |
|  | 1 | 1 | 1 | 1 |  |  | 1 | 1 | 1 | 1 |
|  |  | ? |  |  |  |  |  | ? | 1 |  |

Figure 9‑3: Cases 149442 and 18319 respectively, the only two cases in the top 50 most used that were not created in the first 10 thousand games

This shows a limitation behind the CBR approach. Similar to annealing in other learning methods, it is easy to be selected and used early but as time goes on and win-rates stabilise, any somewhat new case which is selected but accidentally triggers a mine has their win-rates deducted by an often larger amount as compared to an already well used case, which may result in this case to achieving ‘redemption’, as outlined earlier, only after a very long time, or even never.

Strangely enough, case 18442 (on the right in figure 9-3) also shows us that despite not having an out of bounds token, my CBR can detect some board edges, as seen here by the fact that the zeroes seemingly just stop without a defined edge, a side effect of out of bounds cells not having alterable values and of the zero-flush implementation, an emergent interaction which I was not aware of.

# Conclusion

Overall all the requirements outlined have been met, I have conducted extensive tests on them and I have learnt about many aspects from multiple fields in computer science from database and algorithm optimisation, java implementation and of course machine learning and artificial intelligence.

But it appears that due to the lack of conclusive information I cannot say for certain whether or not my hypothesis was indeed correct, but the most likely outcome is that a pure CBR approach with no other injected information is an intriguing learner method but will probably continue to have dubious win-rates. This is due to the sheer number of possible cases even a casebase of half a million cases, it is still only a small fraction of that number with any kind of increase will make the CBR learn slower and slower.

I however did learn many things about the nature of Minesweeper and CBRs themselves. There are still many things left to try in regards to a case based reasoning Minesweeper learner; as I explained earlier the edge-detection CBR served to eventually sort the casebase as cases reached their true win-rate over constant use. This shows that by sorting the casebase further one could increase the speed of the CBR even further, overcoming some of these physical barriers. This edge detection method could be combined with the forgetful CBR to possibly minimise the introduction of false-positives and hopefully allow it to reach this critical mass.

For my other CBRs I do believe that the easiest and probably quickest ways to increase the win-rate would be to introduce more information in the form of play styles or guide lines for the CBR to arrive at more intelligent solutions quicker and to also have a more structured and/or sorted database. For instance; by implementing the right support for an out-of-bounds token, such as a more split database for cases with and without OOB tokens, it could increase the information inside the cases while dampening the problems and impacts caused and outlined earlier in this report.

Another way of improving the CBR is to change the way we think about cases, to shrink the effective maximum amount of cases by equating cases to each other as sets could shrink the casebase to a more potent set of cases for the CBR to scan through, making each good case a more powerful tool, bad cases become unused quicker and reduce the time it took to play a game effectively increasing its growth rate.

Alternatively on the edge-detection CBR a further adjustment could be to change how the scoring system works, having some cells be worth more points than others, effectively changing the sorting order. Revealed cells directly around cell in question could be worth more as having these revealed in conjunction with other cells can provide a lot of information. To this end the implementation of a neural network with augmenting topologies which scanned subsections of boards similarly to my CBR seems to look promising.

While I would have liked to perform more tests and implemented more versions of my CBR I believe I have explored much of this area and have found that while a traditional CBR showed promise, due to its limitations as a learning algorithm, it serves more as a platform for stronger implementations. Of which there are plenty of avenues to explore.

# References

Aamodt and Plaza (n.d.). *Problem Solving By Case-based Reasoning*. [image] Available at: https://www.uni-trier.de/fileadmin/\_processed\_/csm\_Figure\_CaseBasedReasoning\_04\_7a1c5356e5.png [Accessed 2 May 2017].

Castillo, L. and Wrobel, S. (2003). Learning Minesweeper with Multirelational Learning. In: *Eighteenth International Joint Conference on Artificial Intelligence*. [online] Available at: http://www.minesweeper.info/articles/MultirelationalMinesweeper.pdf [Accessed 3 Oct. 2016].

Gardea, L., Koontz, G. and Silva, R. (2015). *Training a Minesweeper Solver*. 1st ed. [ebook] pp.3-5. Available at: http://cs229.stanford.edu/proj2015/372\_report.pdf [Accessed 3 Oct. 2016].

Jpab.org. (2016). *ObjectDB Database Performance - the JPA Benchmark*. [online] Available at: http://www.jpab.org/ObjectDB.html [Accessed 10 Jan. 2017].

Kaye, R. (2000). *Minesweeper is NP-Complete*. 1st ed. [ebook] New York: The Mathematical Insider. Available at: http://simon.bailey.at/random/kaye.minesweeper.pdf [Accessed 3 Oct. 2016].

Quartetti, C. (1998). *Genetic Algorithms and Genetic Programming at Stanford 1998*. 1st ed. Stanford, California, USA, pp.137-146.

Rhee, S. (2000). *Genetic Algorithms and Genetic Programming at Stanford 2000*. 1st ed. Stanford, California, USA, pp.312-318.

Roos, D. (2017). *Minesweepr :: mrgris.com*. [online] Mrgris.com. Available at: https://mrgris.com/projects/minesweepr/ [Accessed 10 Feb. 2017].

Schank, R. (1983). *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. 1st ed. Cambridge [u.a.]: Cambridge Univ. Press.

Stanley, K. and Miikkulainen, R. (2002). *Evolving Neural Networks through Augmenting Topologies*. 1st ed. [ebook] Austin, Texas, USA: The MIT Press Journals. Available at: http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf [Accessed 13 Apr. 2017].