Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Project

Algorithms for Minesweeper Game Grid Generation

Jan Cicvárek

Supervisor: MSc. Štěpán Kopřiva, MSc.

Study Programme: Open Informatics

Field of Study: Computer and Information Science

May 25, 2017

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Jan C i c v á r e k

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Algorithms for Minesweeper Game Grid Generation

Guidelines:

1. Study the game of minesweeper, problem definition and complexity.
2. Study the constraint satisfaction problem and other relevant techniques.
3. Formalize the problem of solving the game and generating the game grid.
4. Propose an algorithm for solving the game when solvable, with emphasis on CPU time.
5. Implement the algorithm described above.
6. Evaluate the algorithm on different game instances.
7. Adapt the algorithm for gradual generation of solvable grid.
8. Evaluate the algorithm on different mine densities and grid dimensions.

Bibliography/Sources:

[1] Stuart Russel, Peter Norvig – Artificial Intelligence: A modern approach, 2nd edition - 2003
[2] Richard Kaye - Infinite versions of minesweeper are Turing complete - Birmingham, 2007
[3] Ken Bayer, Josh Snyder and Berthe Y. Choueiry - An Interactive Constraint-Based Approach to Minesweeper . - University of Nebraska-Lincoln, 2006

Bachelor Project Supervisor: MSc. Štěpán Kopřiva, MSc.

Valid until: the end of the summer semester of academic year 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 14, 2015

Abstrakt

Minesweeper je videohra z roku 1990. Nalezení řešení jedné její instance nebo důkaz jeho neexistence je NP úplný problém. V této práci prozkoumám algoritmy, které tento problém řeší v polynomiálním nebo exponenciálním čase s různou úspěšností. Implementuji svůj vlastní algoritmus s důrazem na vysokou úspěšnost a využitelnost při generování pole. Nakonec také implementuji algoritmus, který je schopný generovat pole hry minesweeper, které je vždy řešitelné a zavedu nové hodnocení obtížnosti, které tento algoritmus využívá.

NP úplné a NP těžké problémy jsou velmi frekventované, lze se s nimi setkat při zajišťování kybernetické bezpečnosti, vývoji nových léků, alokaci zdrojů nebo například při obecném prohledávání stavového prostoru. Hodně NP problémů lze řešit pomocí algoritmů s polynomiální složitostí, které je řeší s vysokou úspěšností, ale nikomu se nepodařilo dokázat, že lze NP problémy v polynomiálním čase vyřešit deterministickým automatem nebo naopak možnost řešení deterministicky v polynomiálním čase vyloučit, proto je každé jejich studium přínosné.

Abstract

Minesweeper is a videogame, first introduced in the year 1990. To find a solution for one instance of this game, or prove that it does not exist, is an NP-Complete problem. In this thesis, I will introduce algorithms that can solve this problem in either polynomial or exponential time with varying success rates. I will implement my own solver, with emphasis on high success rate and its possible utility in gradual generation of the minefield. I will also implement an algorithm that can generate a minefield that is always solvable and introduce new difficulty rating system that can also be used as an input for this new minefield generator.

NP-Complete and NP-Hard problems are very common. We need to solve them in cyber-security, when developing new medicine, optimizing resource allocation, or just when searching a statespace. There are many algorithms that can search for solutions to the NP problems in polynomial time with high success rate, but the NP problems have never been proved to be solvable with deterministic automata in polynomial time or proven to be unsolvable in polynomial time with deterministic approach. For that reason, any study can help us understand these problems better.

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Author statement for undergraduate thesis:

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date…… ……

…….
signature

Acknowledgements

I would like to express my sincere gratitude to my advisor MSc. Štěpán Kopřiva, MSc. for the support of my project, invaluable advice and input and Renata Fialová, Jana Zichová and Ivana Býmová for their amazing support and care that they dedicate to each of their students.

Contents

List of Figures

1 Introduction

Minesweeper is a fascinating computer game. It borrows its mechanics from the earliest computer games, uses a ruleset changed and modified dozens of times to overcome the limitations that machines of that time presented and was ultimately created as a part of a programming exercise of a newly hired Microsoft developer. It would have probably been put away and forgotten, if it wasn't developed during the rise of graphical operating system and when Microsoft was looking for games to include in its first release of Windows Entertainment Pack(WEP), Minesweeper was one of the 8 games that were selected.
It still took two more years for Minesweeper to gain the recognition it has deserved and became a popular video game bundled with many Windows consumer and enterprise releases, same as KDE and GNOME environments. While Minesweeper is just a simple game with many sessions lasting lest than a minute, it holds a key to one of the greatest question in mathematics. Completely understanding Minesweeper means being able to answer the question whether P=NP?, but also whether NP=co-NP?
The NP-complete and co-NP-complete classes contain many important problems. They are perhaps best known for their use in cryptography, but while an algorithm running in polynomial time would allow us to decrypt some of the encrypted data, there is much more we would be able to do, it would allow for better planning and pathing, allowing us to better spread traffic, find optimal routes, allocate resources better, develop and test better electronics, find new medicaments, but also spread them effectively, predict food shortages and find the best way to distribute supplies.
We don't even need to find a polynomial algorithm to help us with these tasks, it is always possible to find an efficient solutions to some instances of a problem, and since any problem in NP can be reduced to any other problem in NP, it is possible that a graph colouring algorithm will be good at finding a route through multiple points or that an algorithm that determines isomorphism of two graphs will be great at solving Minesweeper boards.
With quantum computers being able to use more and more qubits each year, one more complexity class of Minesweeper variation is becoming increasingly important, the RE. One of the strongest suits of quantum computers is solving large problems with periodic properties, which is a behaviour demonstrated by a large Minesweeper boards.
In this paper I will study and formalize the game of Minesweeper, its variations and their complexities, I will also present and evaluate an algorithm that identifies boards that can be solved without guessing that shows promising success rates. I will use this algorithm to implement and evaluate a Minesweeper board generator that outputs only such boards without compromising their complexity with computational intensity that allows it to be used in real-time applications.

1.1 Goals of the thesis

1. **Study the game of Minesweeper and find relevant techniques for solving a Minesweeper board**
   Minesweeper can be deceptively simple and understanding its complexity is important if we want to find a good way to solve the problems it presents. Many

papers and solutions for Minesweeper have been published, but because of the various approaches one can have to Minesweeper, not all of them are directly relevant to the problems of identifying solvable boards.

2. **Propose and implement Minesweeper solver algorithm that can identify solvable boards**
   My algorithm shows good results in the number of solvable boards found. For my algorithm it was essential to prevent any false positives, meaning any board that is marked as solvable has to be solvable. Being able to find more solvable boards while avoiding false positives is important if we want to keep even the complex boards that are difficult, but possible, to solve.

3. **Propose and implement Minesweeper board generator that outputs only solvable boards**
   While the easier configurations of Minesweeper can have very short sessions, the most complex ones can take much longer. When such a game ends not because the player has made a mistake, but because he was forced to guess and outcome and guessed wrong, the whole game can feel unfair. At the same time, if we remove all the complex boards and leave only the one easily solvable, the game could become very simplistic and repetitive. My algorithm is able to generate solvable boards that demonstrate higher complexities than previous solutions.

1.2 Structure of the thesis

This paper is divided into 9 sections:

- 1 **Introduction**
  introduces the goals of this thesis as well as it's motivation and describing its structure.

- 2 **Minesweeper game**
  introduces the game of Minesweeper even for readers, who have never experienced it before with emphasis on technical approach. Those who have prior knowledge of Minesweepers are still encouraged to read through its sections to make sure they are familiar with all of Minesweeper's aspects investigated in this paper.

- 3 **State of the art**
  studies work important to Minesweeper that has been done in the past. It general solving approaches as well as well as concrete solver and generator algorithms. This section also explores the publications that have shown aspects of Minesweeper to be NP-complete, co-NP-complete, co-RE-complete and Sharp-P-Complete as well as how were these results achieved.

- 4 **Technical background**
  goes through relevant methods that help with adapting the CPS approach to Minesweeper.

- 5 **Problem definition**
  defines Minesweeper in technical terms and also formalizes one step in Minesweeper as a constraint satisfaction problem. The overview of complexity of playing a game of Minesweeper is visited.
- 6 **Minesweeper solver & board generator algorithms**
  describes the ideas behind both the Minesweeper solver and board generator. It contains pseudocodes, descriptions of data storage solutions, and approaches that can solve the Minesweeper board and can also be used to generate a solvable board while never getting into an infinite cycle and terminating predictably fast in order to be usable in real time setting.
- 7 **Implementation**
  describes how the ideas from previous section (6) were implemented and how is the application structured. It also contains an UML diagram of the whole application.
- 8 **Evaluation**
  compares the performance of my solver to 3 very different solvers and evaluates CPU time, success rates and also how much are the inadmissible heuristic used in my algorithm. For the generator, it focuses on response time and real-time viability.
- 9 **Conclusion**
  provides a recapitulation of the entire thesis, shortly describes the solution used and if the goals of this thesis were met. It also proposes future work and possible improvements.

2 Minesweeper game

Minesweeper is a puzzle-solving single-player video game. The original version of Minesweeper, originally called Mine was developed between the years of 1989 and 1990 by Robert Donner and based on Curt Johnson's code.[4] The game was inspired[4] by a game published in 1983 called Mined-Out and developed by Ian Andrew.
At the start of the game, the player is presented with a board filled with squares. The player can interact with a board by clicking on the squares, which is an action called probing. Each square hides a number underneath that can be revealed after it is probed. This number describes the amount of mines in the vicinity of the square and ranges from 0 to 8. Some squares also hide a mine. When a square with a mine is probed, the game is lost.
The goal of the game is to probe all the squares that don't contain a mine.

2.1 An example of Minesweeper board with coordinates for each square

During the game, player is provided with a board that can be interacted with. A blue square can be either probed which will reveal a number underneath if it's safe or end the

game if it contains a mine. A blue square can also be marked by the player, which will make it red. Throughout this paper, I also sometimes use letter M to denote a square containing a mine, mostly when I need to paint it in a different colour for demonstration purposes or when the square is done entirely in ASCII.

Figure 1 shows such a board. Along with this board, player will be also shown a number. When no squares are marked, this number will show the amount of mines on the board, where each can contain at most one mine. For each marked square, the number will be decreased by 1, regardless if the marked square is safe or contains a mine.

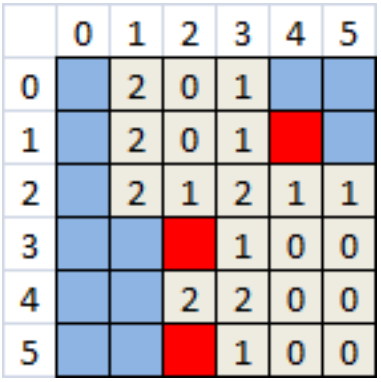

Figure 1: Basic board with coordinates

Figure 1 also has a coordinate system shown by the numbers on white background. This is not a part of a Minesweeper game, but I will use this mapping throughout this whole paper, when referencing a specific square.
There are three standard sizes for Minesweeper board. The **Beginner** board with 9 columns and rows and 10 mines, the **Intermediate** board with 16 columns and rows and 40 mines and the **Advanced** board *(sometimes also called Expert)* with 30 columns, 16 rows and 99 mines[4]. The board on figure 1 is not a standard board, same as most board in this paper, since many of them serve for demonstrative purposes, but they could still be played out, as long as the number of mines is provided.

2.2 Basic parameters of a Minesweeper game

A new board is generated based on these parameters:

$$n, m \in N$$

$$n, m \geq 9$$

$$B \in < 1;\ m * n >$$

$$S = \{s_{0,0}, .., s_{0,m-1}, .., s_{2,m-1}, .., s_{n-1,m-1}\}$$

$$s_{i,j} \in < 0;\ 1 >$$

$$startX \in < 0;\ n-1 >$$

$$startY \in < 0;\ m-1 >$$

$$start \in [startX, startY]$$

Where n is the number of squares in each row of the board, m is the number of squares in each column, B is the number of mines the board contains and S is a set containing objects that each correspond to one square on a board and each square has a corresponding object in this set, $s_{i,j}$ is an object that has the value of 1 if the corresponding square on the board hold the mine $\sum_{i,j} s_{i,j} = B$, $start$ is the position of the square that will be probed first.

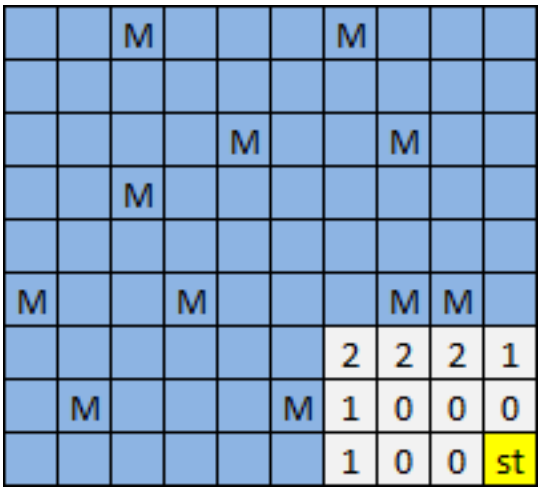

Figure 2: An example board with n=10, m=9, B=11 and start=[9,8] after initialization of the game

The need for the dimensions and number of mines is self explanatory, but asking for starting position might look unnecessary.

The board in figure 3 is the very similar to the one in figure 2, but now start=[0,8], instead of [9,8].

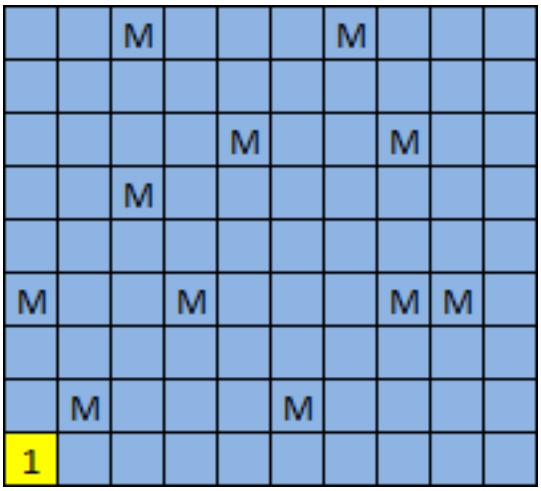

Figure 3: Board from figure 2, but with a different starting position

When the game is initiated, the player is provided with an information that among the squares [0,7],[1,7] and [1,8] is one mine, which is not enough to play the game without guessing and the whole board becomes unsolvable.

2.3 The meaning of numbers displayed on a board and the neighbouring squares

The numbers on the board aren't parameters. They are determined by the game engine and displayed when the player uncovers their corresponding square. These numbers are equal to a number of neighbouring squares that contain a mine, regardless whether those squares are unknown or marked. Squares $s_{[i_1,j_1]}$ and $s_{[i_2,j_2]}$ are **neighbours** if and only if $|i_1 - i_2| <= 1$, $|j_1 - j_2| <= 1$ and $|(i_1 - i_2)| + |(j_1 - j_2)| \neq 0$.

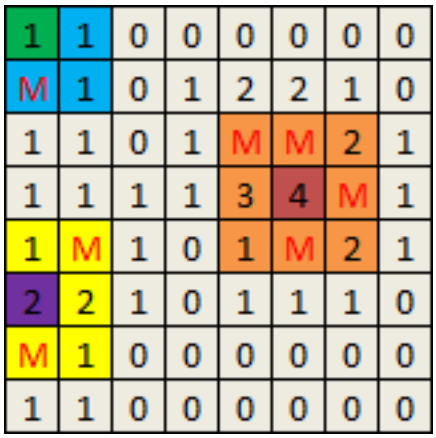

Figure 4: Example of neighbouring squares

In figure 4, we can see a **fully revealed** board with multiple coloured areas, each representing a different example. The green square has **three** neighbouring squares, which are marked with blue colour. The green square also displays the number one. This means that one of the blue squares holds a mine. The violet square has **five** neighbouring squares, signified by the yellow colour. The number two corresponds to the two mines present among the yellow squares. And lastly the brown square with the number four has **eight** neighbours all in orange. Again the four corresponds to the four mines in its neighbourhood.
No two squares have identical neighbours, but when two squares are neighbours, they always share some of their other neighbours. It is possible that a neighbourhood of one square contains all squares in the neighbourhood of another square.

2.4 An example of Minesweeper game-play manually simulated step by step

At the start of the game, the player is presented with fully covered Minesweeper board and the number of mines it contains.

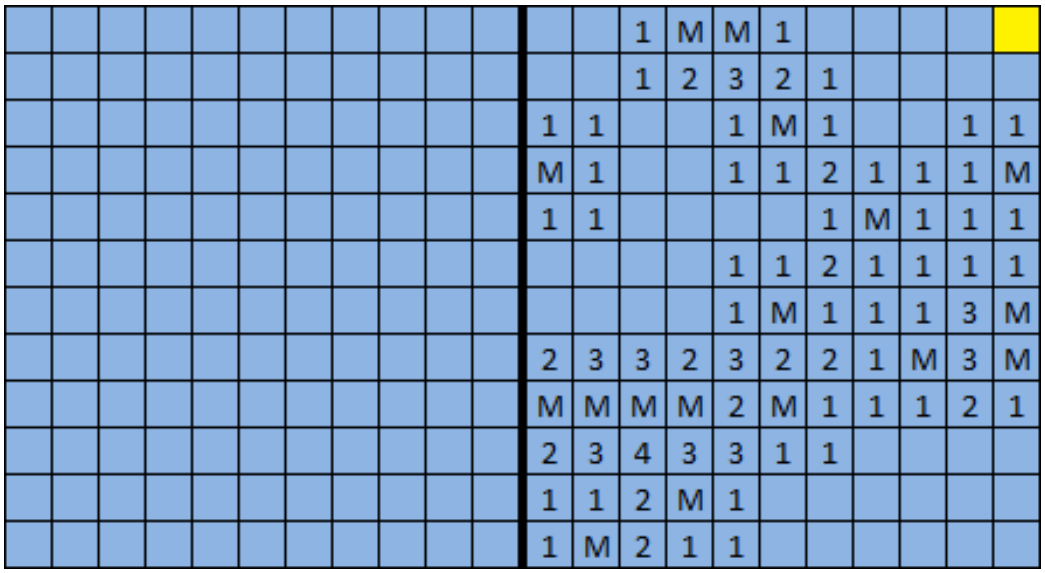

Figure 5: On the left is what the player sees, on the right is the actual board once the starting position is selected

The first step (5) is inherently uninformed, which is the reason why the official rules of Minesweeper[4] state that should the player attempt to probe a mine as his first step, the mine will be moved to the top left corner. If this square already has a mine or is the starting position, the mine will be placed on the first safe unprobed square to the right of the top left corner.

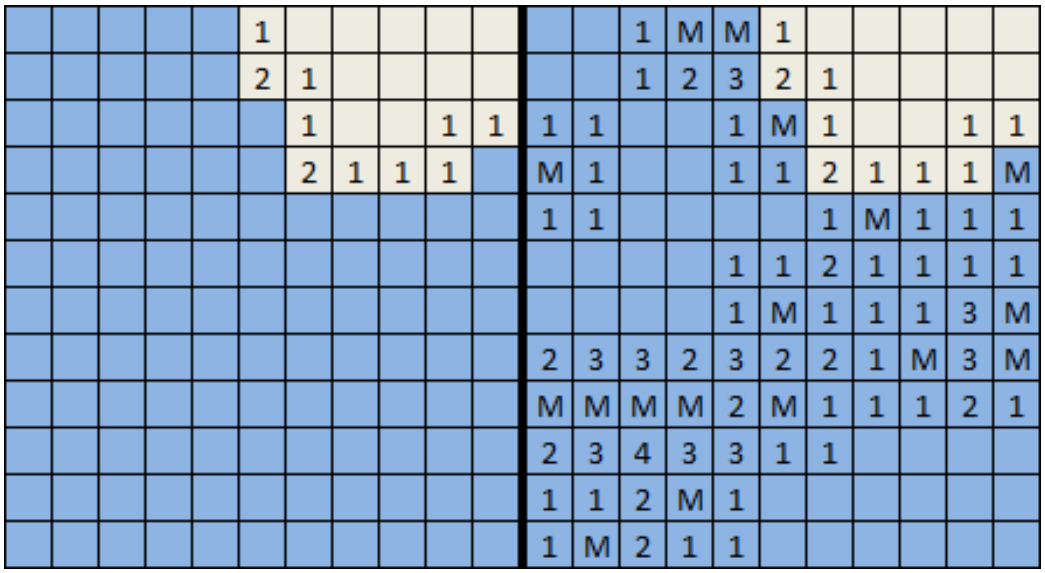

Figure 6: If the first square contains a zero, larger area is unlocked

If the first uncovered square is a 0, the player has a good chance of getting enough information to make a safe move, if it is any non-zero number (6) , the move will have to be an informed guess.

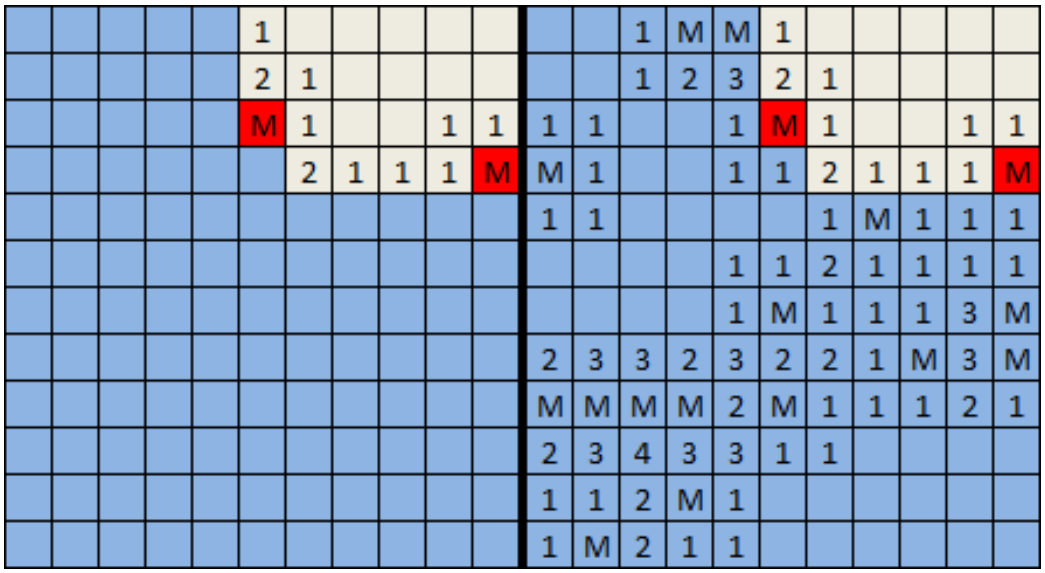

Figure 7: Mines that can be identified

In our example, we had enough information to unveil two mines, thanks to revealed squares [6,1] and [9,2] that only had one unrevealed neighbour and also one undiscovered mine in the vicinity. This is similar to one of the two main steps of SPS described in section 3.1

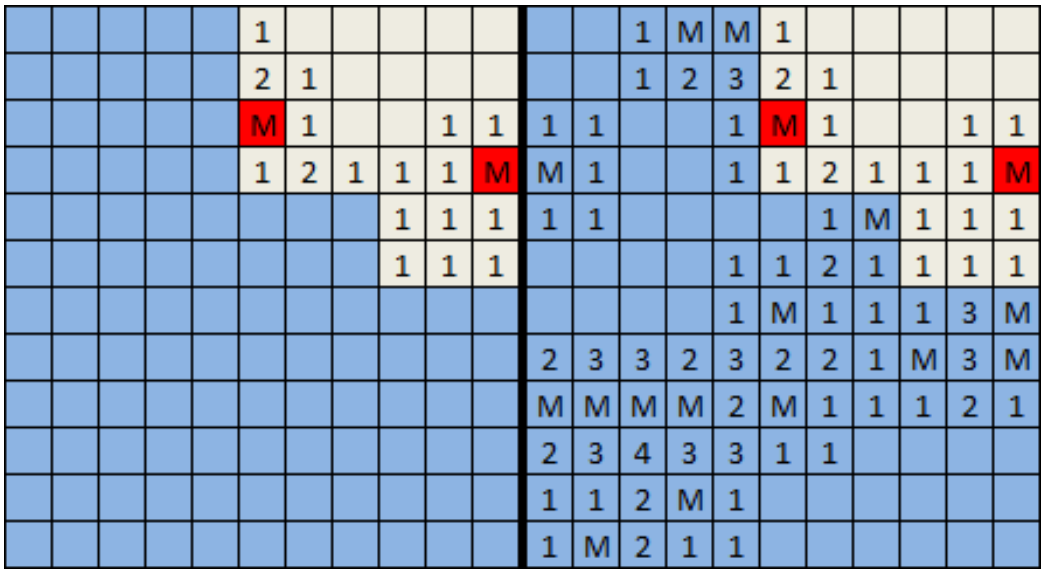

Figure 8: Squares with no mines left

This leaves us with several revealed squares that have all their neighbour mines marked, but still have unrevealed squares in vicinity. These are squares such as [6,2] and [9,3] at first or [9,4] once the neighbours of the firstly named are revealed.

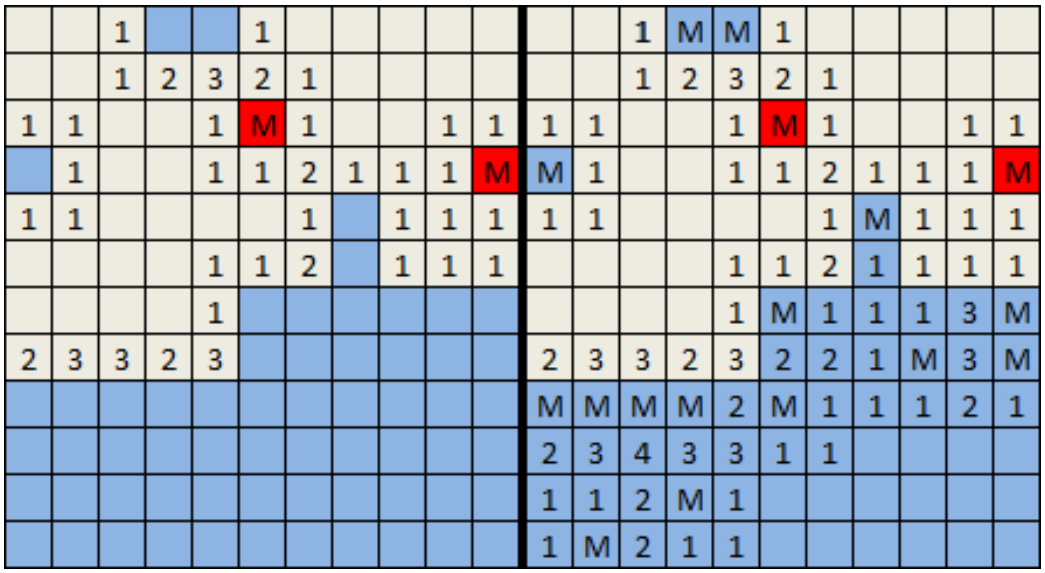

Figure 9: Unlocking the second open area

In the previous paragraph, I have intentionally refrained from mentioning [5,3], since it highlights an interesting situation. By probing every unrevealed square in its vicinity, we chance upon more 0 squares which start a chain reaction unlocking a large portion of the board.

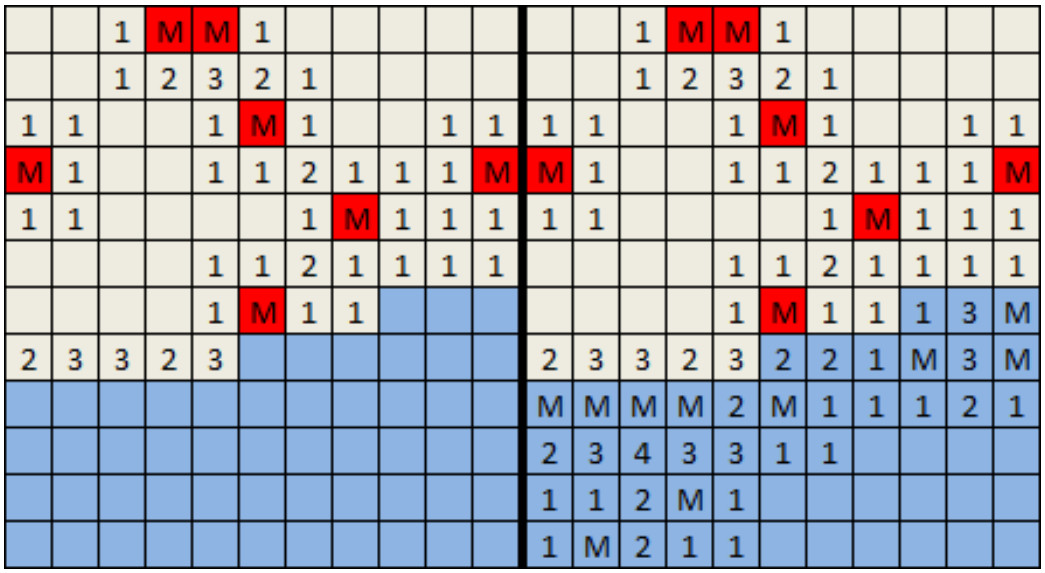

Figure 10: Additional steps

In the next two steps, player can keep unlocking the board, point of interest being the square [6,5] that shows that even squares with higher numbers can often be used to find mines or reveal all neighbouring squares. The next steps can be done applying these rules repeatedly.

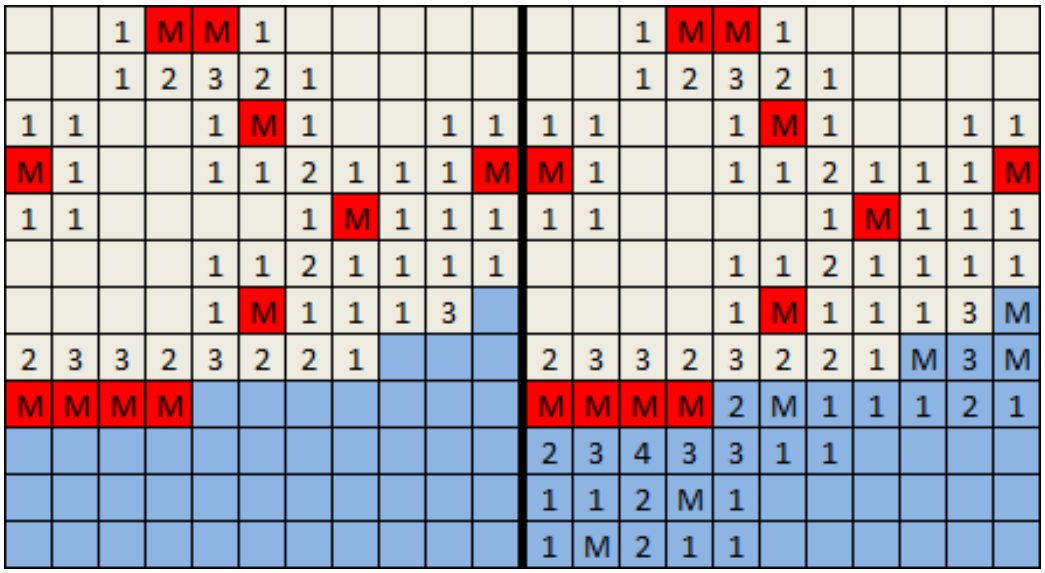

Figure 11: We've used the 1 at [6,6] to uncover its remaining neighbours

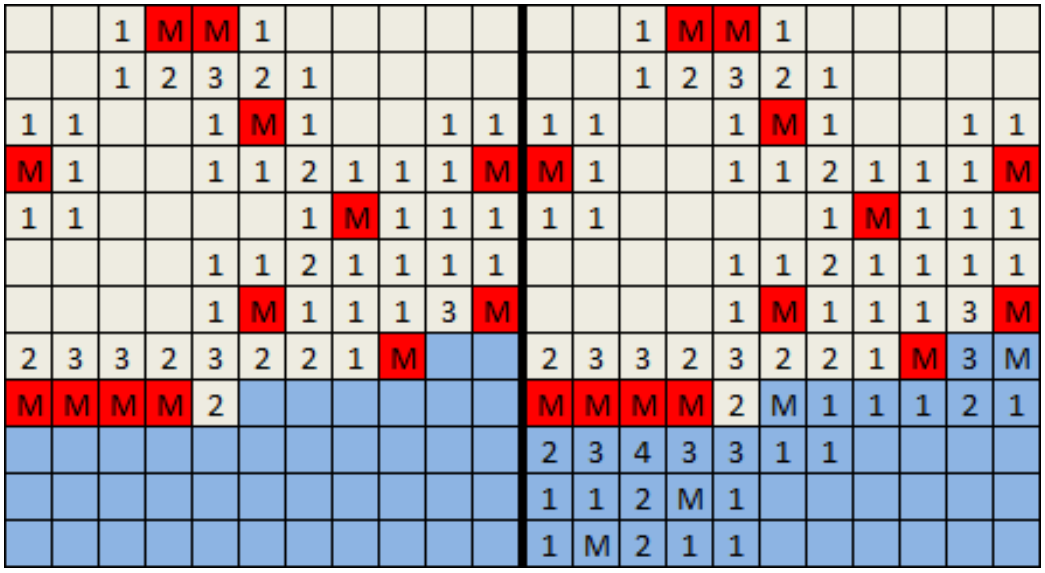

Figure 12: The 2 at [3,7] already had 2 mines, so we can reveal one remaining neighbour

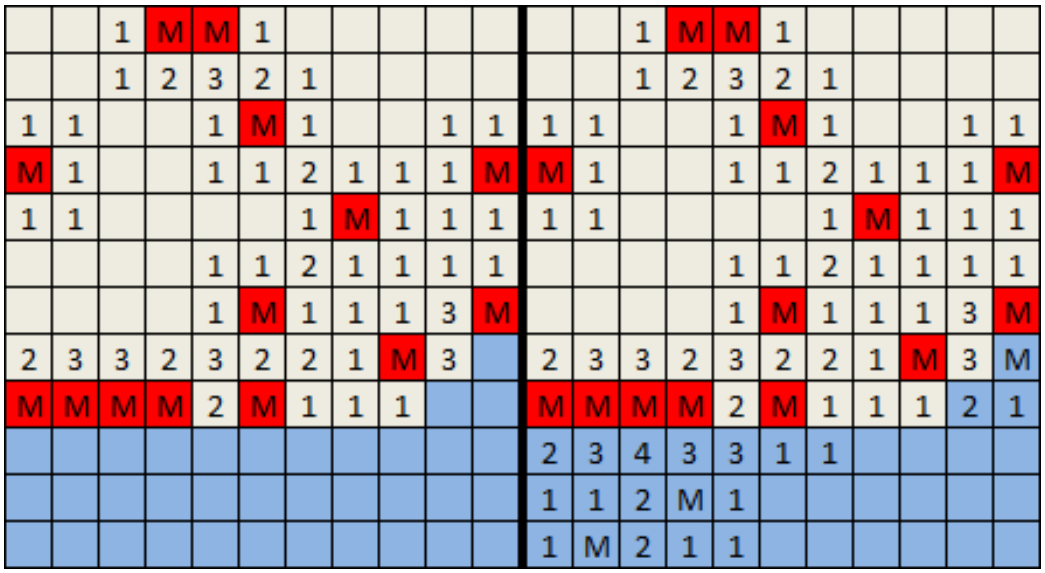

Figure 13: The 3 at [4,7] had only one neighbour and was still missing one mine, while the 1 at [7,7] already had one mine next to it

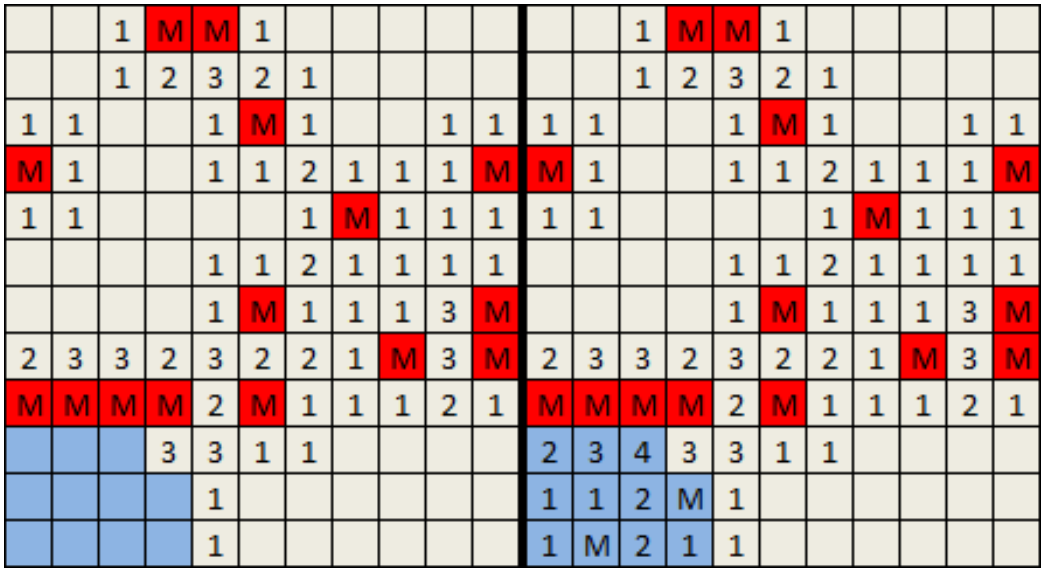

Figure 14: Either of the ones at [6,8], [7,8], [8,8] Unlocking the one remaining open section

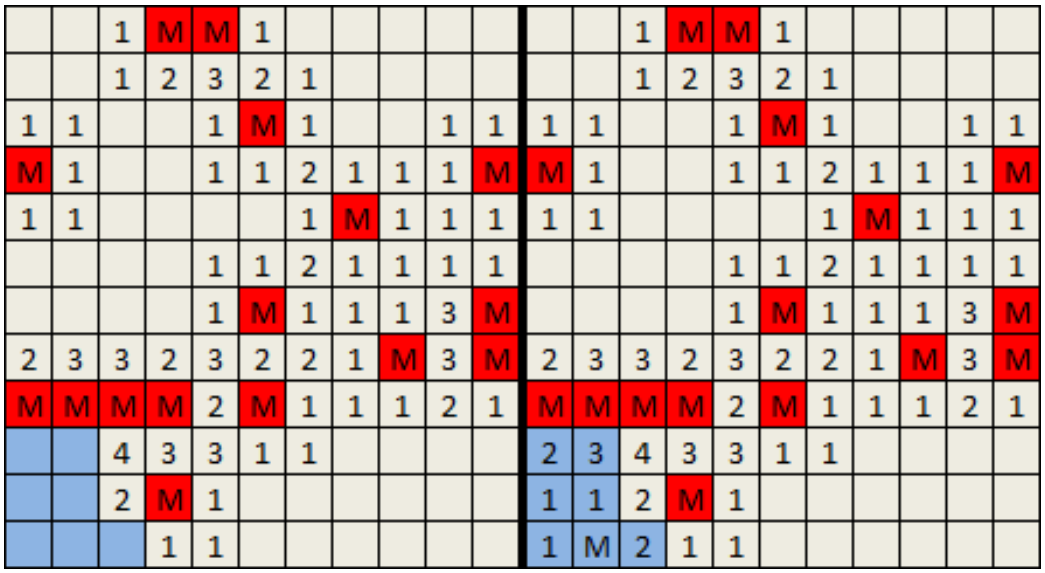

Figure 15: The 3 at [4,9] allows us to identify a mine at [3,10] which completes the correct amount of mines for 3 at [3,9] and 1 at [4,11]

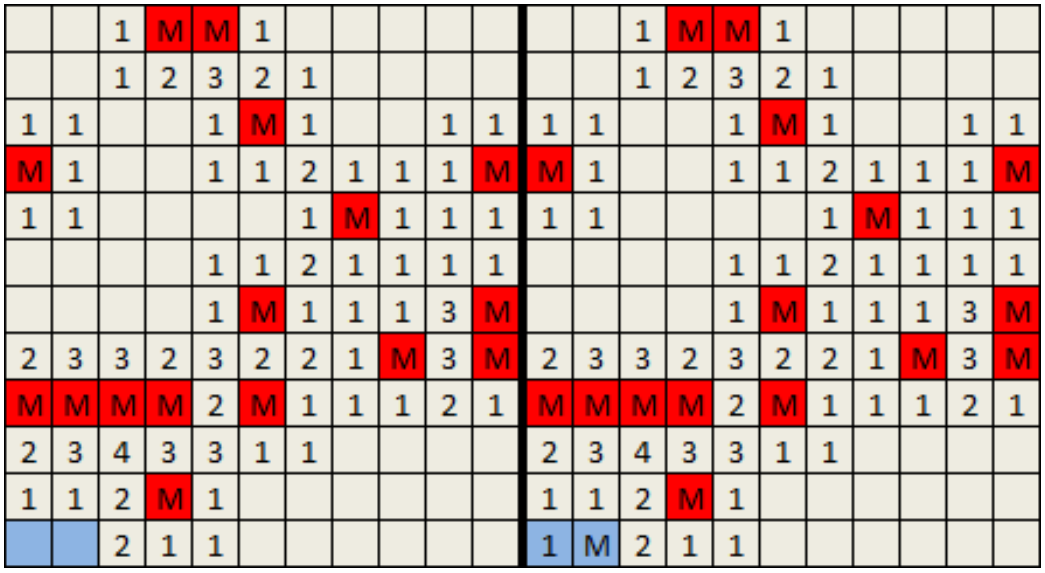

Figure 16: 1 at [3,11] and 4 at[2,9] can be used to reveal additional safe squares and same can be done with 3 at [1,9] after that

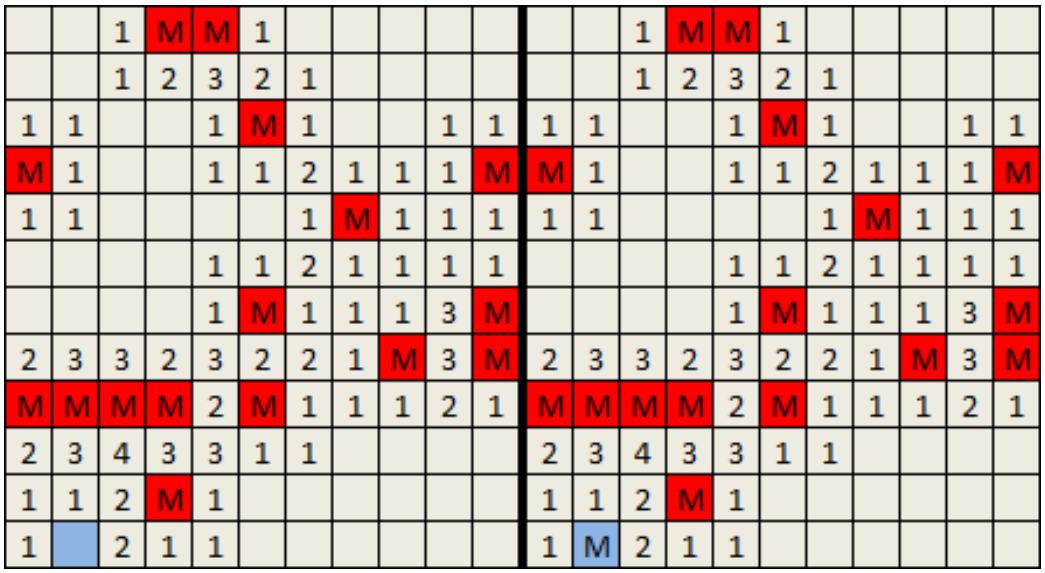

Figure 17: The last step, the board is solved

The last mine didn't have to be marked in order for the board to be solved. Revealing a mine doesn't provide player with any additional information and only helps with orienting on the boards.
In this case, neither an advanced method nor the overall minecount was needed to solve this board, but even a simple board like this cannot be solved from every starting point. Only the squares that are parts of the open sections provide enough initial information to allow the player to make a second step.

2.5 Solvable and unsolvable Minesweeper boards

The game of Minesweeper is considered unsolvable when in any point during the resolution process the amount of information available to the player isn't sufficient to determine any move that might not end in an loss game state.
There are several distinct situations that leave to this state and the difference among them is important to take in account when optimizing the board generator.

2.5.1 Small area of the board is unsolvable - craps shoot

This situation was named by Chris Studholme as a craps shoot.[6] It is not an unsolvable situation, since safe moves are still available, but we can already prove that an unsolvable situation will occur further down the resolution process. We can also make this prediction without the need to consider possible outcomes of the resolution process in other parts of the board.

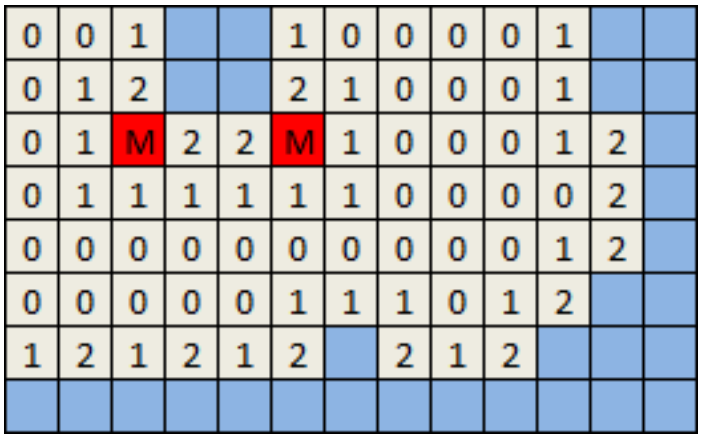

Figure 18: An example of a craps shoot

Even though the figure 18 offers many safe moves, attempting to solve the board will ultimately involve at least one unsafe move where guessing will be required. The unsolvable part in this case is on the top left of the board and only has 4 unknown squares, but no move made in the area spanning from bottom left corner, thru the bottom right to the top right will change situation of the unsolvable top left area.
The whole top left area is a separate cluster (see 6.1.7) that cannot gain any new nodes or constrains (see 6.1.3). Because of this, we will not get any new information on the current unknown squares. And since we already know the number of mines this area contains - two, the total mine count will not help us either.
These situations are easy to find and can be recognized early on in the resolution process and dealt with accordingly to the current task.

2.5.2 Small area of the board is unsolvable, but isn't a craps shoot

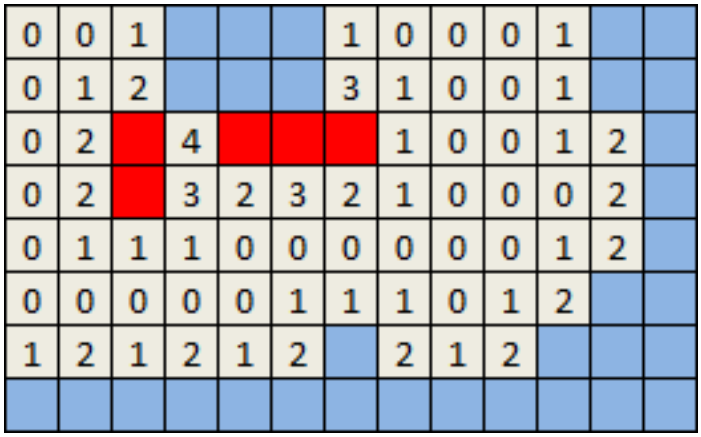

Figure 19: A small unsolvable area that is not a craps shoot

Figure 19 resembles the figure 18 quite closely, but unlike figure 18, it is not a craps shoot.
It would also end up being unsolvable, but we need to consider outcomes of resolving the rest of the board to find out. While in this case, solving the other parts also won't reveal any additional constraints, it will give us the number of mines that are hidden behind the unknown squares on the top left.
The result will be 2, 3 or 4 mines.

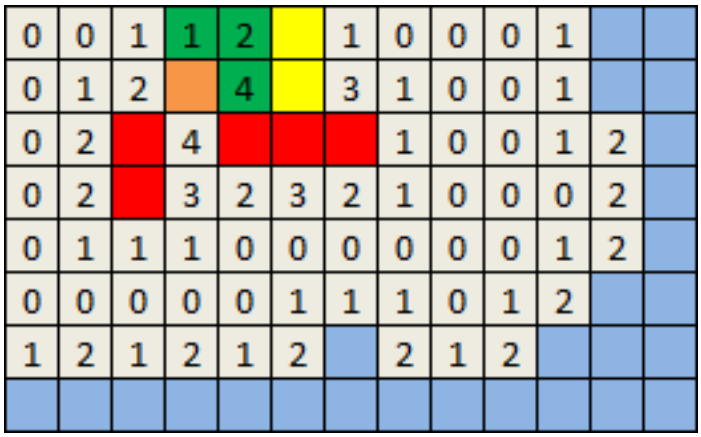

Figure 20: The best case scenario is still unsolvable

If the number of mines is 2, as seen in figure 20, we can safely say that the orange

square contains a mine and we can also reveal the green squares, but the yellow squares [5,0] and [5,1] still have one mine between then and it's position cannot be determined. If the number of mines increases to 3 or 4, the situation will only get worse.

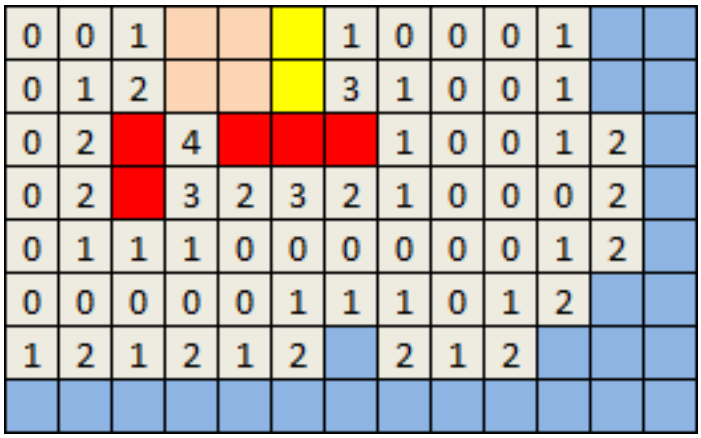

Figure 21: The situation with 3 undiscovered mines

Figure 21 is a situation we will get if the number of undiscovered mines in the upper left cluster is 3. In this case we can make no move at all, since we only know that there is one mine in the yellow area and two in the ochre area, and those two are never next to one another, but across each other.

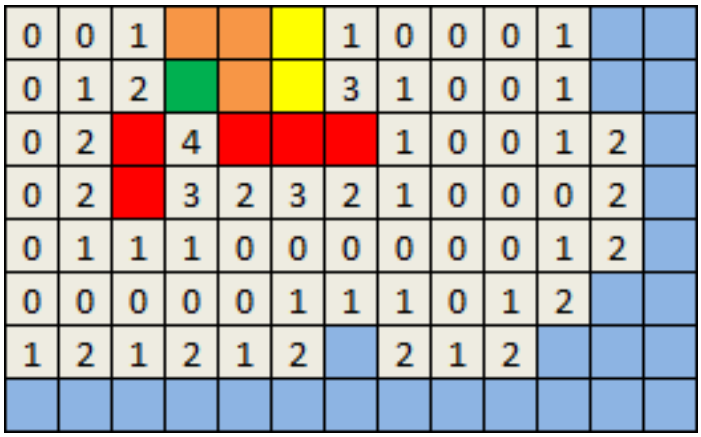

Figure 22: The situation with 4 undiscovered mines

The situation with 4 mines in the upper left cluster, as seen in 22 is also unsolvable. We can safely say that the green square is safe and all the 3 orange ones contain a

mine, but none of that gives us any additional information on the yellow squares, and in figure 20, we have seen that even if we did get additional information, it would still be impossible to determine which of the yellow squares holds a mine.
These more complex unsolvable clusters can help skip a great part of a resolution process, since we would immediately know the board is unsolvable. Regrettably, to my best knowledge, there is no other way to detect them other than to try all the possibilities and all the possible results of the resolutions on the other parts of the board, which makes their discovery very difficult.

3 State of the art

In this section, I will introduce basic algorithms used to solve Minesweeper boards as well as specific approaches to both solving and generating a Minesweeper board. I will also talk about papers that study the complexity of solving a Minesweeper board.

3.1 Single point algorithm

This algorithm is very commonly used in complex solutions, it is used by Ramsdell[7] in Equation strategy (see 3.3), Studholme[6] in his implementation of Minesweeper solver (see 3.2) and Tatham[5] in his version of Minesweeper game client (see 3.4).
Single point strategy (SPS) is solving a constraint satisfaction problem, but it doesn't check the arc consistency [8], which make it one of the least CPU intensive solvers, but it also means that if the board has a solution, SPS is not guaranteed to find it.[14]
Each iteration of the algorithm consists of the following two steps:

- For every clear square, the algorithm compares number of mines in vicinity and the number of marked mines in vicinity. If they equal and there are any undecided neighbours, algorithm marks undecided neighbours as clear.
- For every clear square, the algorithm compares number of mines in vicinity and the number of unrevealed neighbour squares. If the two numbers are equal, algorithm marks all undecided neighbour squares as mines.

For each step of this algorithm, we have to go through all of the clear squares. But number of these squares is always below m*n (see 2.2) and computing time needed to resolve each of these squares within one step is constant. Thanks to this, we can make every step in polynomial time.[13] This algorithm terminates when the grid doesn't change after one step. In this case, we have to apply more complex solver, but whenever a new clear square is found, we can go back to single point solver and try to get more steps done in polynomial time. Since we will be using this algorithm in our solver, here is a pseudocode of the implementation:

```
initialization;
while action made in last iteration do
    for i = 0 to Height do
        for j = 0 to Width do
            if clear square then
                if neighbour mines==marked neighbour mines &&
                undecided ≥ 0 then
                    reveal neighbours
                end
                if neighbour undecided==unmarked neighbour mines &&
                undecided ≥ 0 then
                    mark neighbours
                end
            end
        end
    end
end
```

Algorithm 1: Single point strategy

3.2 CSPStrategy

C. Studholme implements a 7 step algorithm[6] where each state of the playing board is implemented as a Constraint Satisfaction problem and the constraints are represented as a set of equations, named CSPStrategy.
These equations are simplified and divided into subsets of equations with same variables. These equations are solved with a backtracking algorithm. Variables are each in turn assigned a value. After every such assignment, all the constraints are checked. If there still is a solution for the equations, another assignment is made, if there isn't one, backtracking fetches previous configuration. When domain of any variable is reduced to only one, the new step is made and new constraints formed.
The CSPStrategy algorithm is able to solve a Minesweeper board faster than any available algorithm and is also able to solve more boards than algorithms that have been introduced in the past. Studholme's results show and mention very similar CPU time requirements for "intermediate" and "expert" games, but my own trials didn't see any such effect and have instead shown exponential growth, which was also confirmed in Stuholme's other test.
It is important to note that Studholme's algorithm would chose to explore unsafe square before finding a safe next step. This occurs in situations, where no other information can be gained on the particular problematic area of the board.

3.3 Equation strategy

The Equation strategy also interprets the game as a set of linear equations[7], but it relates the equations directly to the square on the board they originate from and uses this relation to go through them. The algorithm itself consists of three independent

algorithms that run successively, independently on the result of the previous one, but they all work and modify the same board and set of equations.
The first algorithm is the single point algorithm that was introduced earlier. In the implementation by John D. Ramsdell, this is achieved by looking at the each equation individually and trying to solve it on its own.
The second algorithm doesn't actually make any steps, it just expands the sets of known equations in one special case. Each square has a between 0 and 8 adjacent uncovered squares. These adjacent, or neighbour, squares are often adjacent to more than one clear square, creating an overlap. Assume s1 and s2 are clear squares with non-zero number of uncovered adjacent squares, s1 $\neq$ s2. Let the set of uncovered squares adjacent to s1 be called e1, and the set of squares adjacent to s2 called e2. Every element in e1 is also element of e2. This means that when we create a virtual clear square s3 adjacent to every square in e2 $\setminus$ e1, the value of s3 will be equal to the value of s2 minus the value of s1.

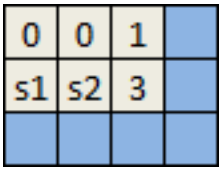

Figure 23: Example of the subset rule used in second algorithm

This will give us a new equation that can be solved for or used in the third algorithm. This approach is often very effective and can produce problems solvable with single point algorithm in polynomial time.
The third algorithm focuses on subtracting two equations from one another. They don't have to be a subset of the other, but they do have to overlap. The algorithm is trying to find an equation where the amount of positive constants is equal to its coefficient. Since all the constants will be 1, 0 or -1 and the variables have to be 0 or 1, this equation can be used to determine the values of all the non-zero variables.
The most basic use of this rule is creating equation with zero coefficient (see Figure 24). For the criteria to be met, the constants in the equation we are subtracting from have to be a subset of the other equation's constants. In Figure 24, we can subtract the equation whose coefficient is at (4,2) from the equation with coefficient at (3,2). The resulting coefficient will be zero, same as the constants at (3,3) and (4,3) and the constant at (5,3) will be equal to -1, which means the corresponding square is clear and can be probed.

\	0	1	2	3	4	5	6
0	0	0	0	0	0	1	
1	0	0	0	0	0	2	
2	1	1	1	1	1	1	
3			2				
4							

Figure 24: Subtraction of equations, basic example

When the two equations aren't subsets of one another, the coefficients need to have different values in order for the algorithm to work. In Figure 25, we can see the board from Figure 24 after the step described above. We can subtract the equation with coefficient at (5,2) from the equation with coefficient at (5,1). The resulting equation will have an coefficient of 1 and the only positive constant at (6,0) and two negative ones at (6,3) and (4,3).

\	0	1	2	3	4	5	6
0	0	0	0	0	0	1	
1	0	0	0	0	0	2	
2	1	1	1	1	1	1	
3			2			3	
4							

Figure 25: Subtraction of equations, advanced example

This algorithm doesn't add new equations to the set, the only equations it produces are the ones that can be directly used to make a step, but some steps can be only determined by combining more than one of the basic equations taken directly from the board. It would be necessary to explore any combination of any number of equations, or in other words, all the combinations on the set of equations with the number of elements going from 2 to the cardinality of the set, which would cause the complexity this algorithm to be factorial. With only two equations at a time, its complexity polynomial, like the other three main rules.
The equation strategy is also able to take into account the absolute number of the mines on the board. This rule only triggers once the number of uncovered squares is lower

than pre-determined value. The threshold was set at 8 in the version I have been studying.
This rule is implemented by simply adding another equation. The coefficient is the number of mines left to discover and the variables are all the uncovered squares. Eight variable is the most any equation not based on this rule can have, and even that would require a guess away from the border of the board and will overlap with much less other equations. Setting a threshold for adding this new equation could certainly highly increase the computation time and it's consistency.

3.4 Simon Tatham's Portable Puzzle Collection

3.4.1 Minesweeper solver

The solver algorithm implemented by Tatham[5] tries to solve the Minesweeper board in stages. It does not guarantee to find solution to every solvable board, but it always finishes in polynomial time.
Since this algorithm is used in a board generator, it isn't just trying to solve the board that is given. It doesn't accept a new board, it can't handle the initialization of the game, but it can accept a board in any stage of a game and attempt to modify it, dividing it into solvable and unsolvable parts.
Apart from these differences, the resolution process is fairly similar to the algorithms mentioned previously. Tathams implementation represents the board as a group of sets, but these sets consist of a known square and it's unknown neighbours, something Studholme and Ramsdell both called equation. At this point in the algorithm, there is no attempt to divide these sets into groups based on their overlapping. At this point a custom version of single point solver gets used repeatedly until it fails to provide result.
Next step is based on the same idea as Ramsdell's equation subtraction rule (see Figure 25). Instead of subtracting two equations, this strategy divides overlapping sets A and B into their conjunction, represented as variables with 0 coefficient in Ramsdell's solution, set of elements in A but not in B, -1 coefficients in Ramsdell's solution and also the elements in B not included in A, having coefficients of 1 in Ramsdell's implementation. While the intersection cannot be solver for in this manner, it is possible that there is only one way to fill the unknown squares in the complement sets, which is the situation this part of algorithm is searching for. If any progress is made by applying this rule, the whole algorithm starts over, otherwise, the algorithm searches for subsets among the sets (see Figure 24). It doesn't aim to create new sets, in this case, the rule just uses the subset to make the other set smaller, basically subtracting.
All these steps have polynomial complexity, but this algorithm can also deal with the more complicated cases. The cost of this action is exponential and because of that, this strategy is only used when there is less than 10 remaining unknown squares. The algorithm explores all the possible configuration of these squares, it does so in a recursive manner, but the recursion itself isn't used to save on function calls and computation time and a list is used instead, to manage the current depth. It is only here that the algorithm divides the sets into groups that don't overlap one another to save on computation time. If this algorithm makes any step, the whole process starts again from the beginning.

3.4.2 Minesweeper board generator

But the solver isn't guaranteed to find a solution, even if there is one. If such a situation arises, the whole board isn't discarded, instead, and algorithm that is trying to make the board easier is called. When it's first called, it relocates the squares that were unknown, but next to known squares. It tries to place them in the clear areas of the board with no mines around. When this fails, it puts them next to known mines.
When the board is still not solvable, all unknown squares are relocated in this fashion. When even this strategy fails, known mines would also get shuffled. Again, the first step tries to level the density, but the second one places them next to each other, creating large mine clusters. This strategy is not guaranteed to produce a solvable board, but it does tend to produce them after enough iterations.
Such a strategy leads to boards that have trends different to those that generated randomly, but have been confirmed as solvable. It eliminates all big empty spaces that can be cleared in one click in a large chain reaction set of by the player. It also tends to place a lot of mines together when the filling up of empty spaces fails and eventually move those groups of mines are placed at the borders of the board, which basically makes the game have less mines but also smaller dimensions. The resulting boards are solvable and require more than average number of steps to complete, but never a complicated deduction process. The big splash square reveals where much of the board clears at once are very rare and sometimes the board is virtually smaller and the borders of it cannot be reached, but the computation time is very low and can provide boards in real time.

3.5 Minesweeper is NP-Complete, the original proof by Richard Kaye

In this paper, Richard Kaye proves that a non-deterministic Turing machine can play a Minesweeper game in polynomial time, that any NP-Complete problem can be reduced into Minesweeper board in polynomial time and also reduces one step in Minesweeper game into known NP-complete problem.[2]
The first part of formal definition of NP-Completeness is shown by proposing a non-deterministic algorithm based on the yes-no problem that is able to solve a Minesweeper board in polynomial time. When given a board consistent with the rules of Minesweeper, it makes a change by marking a blank square and asks if this set-up is still consistent with the Minesweeper rules and information on the board, in case the answer is negative, the square is safe and can be probed.
Next, Kaye shows how one step of Minesweeper game can be reduced to Boolean Satisfiability Problem. This method can be used to describe the whole board, but requires up to 90 inputs for each square and in a advanced games of Minesweeper, tens of squares need to be often considered at once. With this in mind, the most successful sat algorithms are able to solve circuits with over a thousand inputs in less than a second on consumer grade computers[9] and could potentially perform better than the fastest dedicated Minesweeper solvers.
The most important part of this paper is reducing the SAT problem into Minesweeper board that is solvable after the inputs are initialized. This is done by proposing a build-

ing blocks, consisting of wire, 90 degree bend in a wire, a three-way splitter which terminates in one output in the original direction while the other two being perpendicular to it. The wire is made of three square wide blocks and possible alignment problems are solved by introduction of a wire with 8 square wide blocks, making it possible to connect any block.
To be able to construct any Boolean formula, Kaye also shows implementation of the NOT gate and the AND gate. All the other gates can be created from the combination of these two gates. All the wires that carry signal are in horizontal or vertical orientation and can be combined with each other. The wires can also be crossed with the use of three XOR gates, where each can be created by 4 AND and NOT gates and a two-way splitter, created by three-way splitter and wire termination.
All the blocks except for the AND gate can be reached in a standard game of Minesweeper that only has one starting point. The wires will split the board into separate sections, but any wire or a 3-way splitter will uncover all the areas adjacent to it. The AND game presented by Kaye can't return an output after after applying the inputs if the whole game was started from just one position. Instead, several areas of the board need to be uncovered first. Even though it can provide an output, the player still can't safely reach this conclusion, since it requires a guess[13].
In publication *Some Minesweeper Configurations*[3], Kaye presents an OR gate that also can't be resolved from a single starting point and XOR gate that has the same problem, but when additional areas are revealed prior to the game start, it can be resolved by the player without guessing after all the inputs are initiated and when we add the already working NOT gate, we can use these gates to create any Boolean formula[15], including the disjunctive normal form, same as conjunctive normal form[16].
In the paper *Infinite versions of Minesweeper are Turing complete*, Kaye proves that the consistency problem on a grid with infinitely many rows and columns can be shown to be inconsistent in a finite ammount of time by a Turing machine and is a co-RE-complete problem[1]. He does this by considering all the possible configurations and finding a periodic pattern in them. This result is unexpected, since an analogous problem (consistency) on a finite grid was shown to be NP-complete and it was the inference problem that was proved as co-NP-complete[17].

3.6 The complexity of Minesweeper

In the publication *The complexity of Minesweeper and strategies for game playing*[13], Kasper Pedersen builds on the proof presented by Kaye[2], proving the NP-completeness of a problem he calls **consistency**. Pedersen defines consistency as a yes/no decision problem that occurs with each state of the board after initiation. It asks whether the current board with its covered and uncovered squares has any mine distribution that agrees with all the information the current state of the board provides (see 2 for general info on Minesweeper boards).
Pedersen shows that *consistency* is in NP by presenting and algorithm that can answer this question in polynomial time on non-deterministic automata. The NP-hardness is also proved by building a logical circuit (see 3.5 for description of Kaye's proof), but Pedersen points out that Kaye's original proof uses AND gate that can be resolved in Minesweeper in various ways, even if the inputs stay the same, although, the gate will

still output the same result. To mitigate this, Pedersen presents his own designs of OR and XOR gates that can alternatively be turned into AND gate with the use of NOT gates[15].
Both Pedersen's and Kaye's proofs were done for the 2 dimensional board, additionally, Pedersen also proves NP-completeness of the higher higher dimensions. A polynomial complexity is also proved for 1-dimensional Minesweeper game, but both 1 and 3 or more dimensional versions doesn't follow the basic rules of Minesweeper[2.2]. Sharp-P-completeness is proved for the task of counting possible assignments that the current state of the board allows, which is important, if the aim is to make a move even with incomplete information. In contrast, this paper only focuses on boards that always contain enough information to allow for a safe move to be found, so as soon as we find a second configuration that is consistent with the information on the board, we can stop searching, since the board is inadmissible.

4 Technical background

In this chapter, I will describe and explain all the well-established methods I have used for my solution. This section doesn't contain the specifics of their implementation, those are found in the sections 6 and 7.

4.1 The constraint satisfaction problems

When a problem can be described as a set of variables that can take certain values and a set of constraints on the values those variables take, we can solve it as a constraint satisfaction problem, or CSP for short. Rather than devising a different algorithm for the problem, we can define it as a CSP and use a general CSP solver that can solve any problem defined in this way.

The CSP is defined by three components[10]:$< X, D, C >$
X is a set of variables, $\{X_1, X_n\}$
D is a set of domains, $\{D_1, D_n\}$
C is a set of constraints, $\{C_1, C_m\}$
Each variable $X_i \in X$ has its domain $D_i \in D$ that determines the values it can take. The values any variable can take are further limited by the constraints. Each constraint C_i involves a subset of variables and specifies the allowable combination of values for that subset.[9]
In general a CSP solver works by setting values for the variables from their domains and checking whether the assignment doesn't violate any constraint. If it doesn't, the assignment is considered to be legal. If all the variables haven't been given a value, the assignment is considered to be partial. If all the variables have a value and the assignment is legal, it is also complete.

4.1.1 Minimum remaining values heuristic

Every solver for CSP has to pick a variable and decrease the size of its set of domains. This choice doesn't have to be random. The minimum remaining values heuristic, or MRV for short is one of the most successful in variety of applications.[12]
When the algorithm has to decide what domain to reduce next, it chooses the smallest one. This is applicable even for variables that are only left with one member in the set of domains. This approach helps reduce the branching factor, because when we are presented with less choice, we can branch out less.[10]

4.1.2 Degree heuristic

The degree heuristic is often implemented on top of MRV. In cases when MRV chooses more than one candidate, it selects the one to be explored. Naturally, it can also be used on its own.
It chooses variable that shares constraints with greatest number of unassigned variables[10], the goal is similar to the MRV, we are trying to limit the number of choices the algorithm has to explore.

4.1.3 Most constraining value strategy

This method would increase the branching factor and computing time in most CSPs, since it does the exact opposite of lest constraining value heuristic. It's goal is to force the algorithm into bad decisions that end in dead ends. This can be advantageous when we aren't concerned about the CPU time needed to get the first complete assignment.
The strategy is used when we need to pick a value for a specific variable. It chooses the value that decreases the domains of other variables the most.
This strategy can find illegal partial assignments quickly and can be useful when searching only a part of a state-space. This strategy is useful only in special cases and in general increases the computation time needed to reach the solution.
It can be very useful tool for disproving assumptions, it can tell us what assignments don't work. In problems with one and only one solution, it can make proving that the complement of the assumption is correct faster.

4.1.4 Forward checking

Forward checking can be called when a value is set for previously unassigned variable.[10] It only works locally, meaning it will check only the variables that share the constraints with the modified variable.[12] This means the computation isn't dependant on the size of the problem, only the size of the constraints on the modified variable. When it is called, it checks whether these constraints can still be satisfied[12], and removes all the values from the domains that would contradict those constraints. It allows us to make better decision and thanks to its low CPU cost, it is usually able to increase the effectiveness of the whole algorithm.[11]

4.1.5 Arc consistency

Arc consistency is essential part of any algorithm designed to solve CSPs. It makes sure that a partial assignment is legal[10]. The arc consistency is usually checked frequently and makes up most of the computation cost of a solver.[11] Frequent arc consistency checking can detect dead ends quickly and makes backtracking easier[10].
When arc consistency algorithm is called, it explores all the variables that have been modified since the last call. It removes all the subsets of values that cannot be reached after the modification and reduces the domains to respect the new constraints. When this results in a variable with domain of size one, it is added to the evaluation. A simple arc consistency algorithm would achieve this by simply running forward checking on all the variables as long as any changes are being made[12]. AC-3 algorithm would only do the forward checking on the variables that have had changes done to their domains since the last time the arc consistency was checked.

5 Problem definition

The goal of this thesis is creating and a describing an algorithm that, when given the basic parameters (see 2.2), generates a Minesweeper board of any complexity (see 5.2 for different scales of complexity in Minesweeper) that can be solved without the need to guess.

The aim of this section is to introduce additional ways to describe Minesweeper and its properties. I present a way to mathematically describe the current state of Minesweeper board as a constraint satisfaction problem along with some redundant variables that can help understanding the board. I also briefly touch on the different complexities a Minesweeper board can have.

5.1 One step in the Minesweeper game as a constraint satisfaction problem

Finding the next step in the game of Minesweeper can be done as a decision based on a results of multiple constraint satisfaction problems (CSP).
First, let's try to define any state in an initiated Minesweeper game as a single CSP.

CPS is defined by three sets, $< X, D, C >$. For a general Minesweeper game, they would be:

$$
\begin{aligned}
&X_1, ..., X_k \in X \\
&i \in < 1;\ k > \\
&w_i \in < 1;\ n > \\
&z_i \in < 1;\ m > \\
&\{\forall X_i \in X | X_i = \{[w_i, z_i]\}\}
\end{aligned}
$$

$D_1, ..., D_l \in D$
$j \in < 1;\ l >$
$\{\forall D_j \in D | D_j = \{0, 1\}\}$

$C_1, ..., C_o \in C$

X contains all the squares that are unknown but are neighbour to at least one known square. **D** contains domains of all of the squares in **X**, where 0 represent a square with no mine and 1 a square containing mine. **C** is a set of all the information we have on the placement of the mines in the squares contained in **X**. Writing them as a equations is a allowed, but most general solvers would not accept this representation.

0	0	0	0	1				0	0	0	0	1	M	2	1
1	1	1	1	2				1	1	1	1	2	2	M	1
								M	1	1	M	1	1	1	1
								2	2	3	3	4	2	1	0
								2	M	3	M	M	M	1	0
								2	M	3	2	3	2	1	0
								1	1	1	0	0	0	0	0

Figure 26: On the left is what a player sees, on the right is revealed board

When we know what the sets, $< X, D, C >$, we can try to define them for a board on figure 26. It might at first look obvious what those sets should be, for example if we are trying to find the next step for the board in figure 26, we could try:

$\{[0, 2], [1, 2], [2, 2], [3, 2], [4, 2], [5, 2], [5, 1], [5, 0]\} = X$
$D_1, ..., D_8 \in D$
$i \in < 1;\ 8 >$
$\{\forall D_i \in D | D_i = \{0, 1\}\}$
$C_1, ..., C_6 \in C$
$\{([0, 2] + [1, 2] = 1),$
$([0, 2] + [1, 2] + [2, 2] = 1),$
$([1, 2] + [2, 2] + [3, 2] = 1),$
$([2, 2] + [3, 2] + [4, 2] = 1),$
$([3, 2] + [4, 2] + [5, 2] + [5, 1] + [5, 0] = 2),$
$([5, 1] + [5, 0] = 1)\} = C$

Unfortunately, the result of CSP defined like this might not be the solution we are looking for.

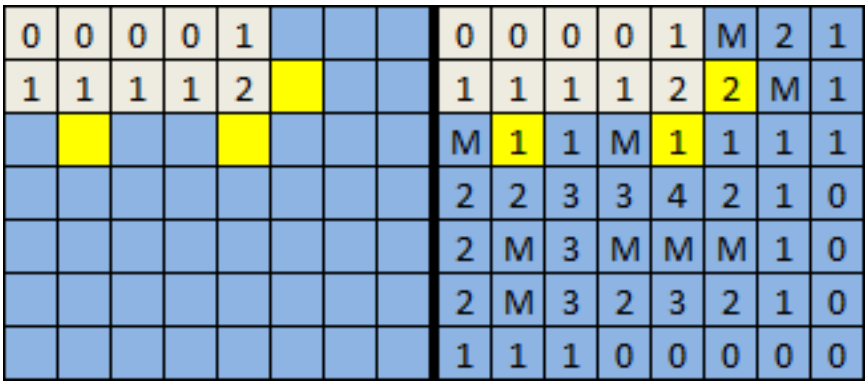

Figure 27: Possible result of basing the next step on one CSP for the whole board

In figure 27, we can see a solution that satisfies all the constraints, and is a possible result produced by CSP. As we can see on the unveiled version of the board, that solution is not correct and basing the next on it would result in losing the game.
What we are really looking for is a square that has to have the same value in any possible complete assignment. This means we want a partial assignment that is part of every possible complete assignment, but this is something a general CSP is not efficient at solving.
What we can do is simulate one step and then check if that is consistent. The set X and C would remain unchanged, we would just reduce one of the domains in D to only contain one value. We would of course need to iterate through all of the domains, but in order to show this strategy having success, let's reduce the domain of [2,2]:

$$D_{[0,2]} = \{0,1\}, D_{[1,2]} = \{0,1\}, D_{[2,2]} = \{1\}, D_{[3,2]} = \{0,1\}, D_{[4,2]} = \{0,1\}$$
$$D_{[5,2]} = \{0,1\}, D_{[5,1]} = \{0,1\}, D_{[5,0]} = \{0,1\}$$
$$D_{[0,2]}, D_{[1,2]}, D_{[2,2]}, D_{[3,2]}, D_{[4,2]}, D_{[5,2]}, D_{[5,1]}, D_{[5,0]} \in D_6$$

The sets **X** and **C** will be unchanged.

0	0	0	0	1	M	2	1
1	1	1	1	2	2	M	1
M	1	1	M	1	1	1	1
2	2	3	3	4	2	1	0
2	M	3	M	M	M	1	0
2	M	3	2	3	2	1	0
1	1	1	0	0	0	0	0

Figure 28: One of 16 possible steps can be explored by running CSP on the sets $< X, D_6, C >$

If we use the CSP on the sets $< X, D_6, C >$, we will get no result. This will prove that there is no arrangement that would allow [2,2] to be mine, which means we are safe to assume it doesn't contain a mine and we can probe it. Even in a simple example like this, As much as 15 CSPs need to be solved before we find a result that would help us.

5.2 Additional parameters of Minesweeper game

There are also some additional parameters that can be deduced from the base parameters. They aren't necessary to initiate the board generator, but can help us to describe various relations, properties, situations and so on.

$$V = \{v_{1,1}, .., v_{1,m}, .., v_{2,m}, .., v_{n,m}\}$$

$$v_{i,j} = \sum_{i_s=i-1}^{i+1} \sum_{j_s=j-1}^{j+1} s_{i_s,j_s}$$

$$3BV \in \{1, .., m * n * (8/9)\}$$

$$Rank \in \{0, .., m * n\}$$

Where V is a set containing objects that each corresponds to one square on a board and each square has a corresponding object in this set, $v_{i,j}$ is an object that has the same value as the number of mines neighbouring the corresponding square.
$3BV$ is a method of Minesweeper board ranking introduced by a collaboration of Stephan Bechtel, Benny Benjamin and other high ranking Minesweeper players[4]. The acronym stands for Bechtel's Board Benchmark Value and the method determines the minimal number of times the player have to probe a square in order to win the game.

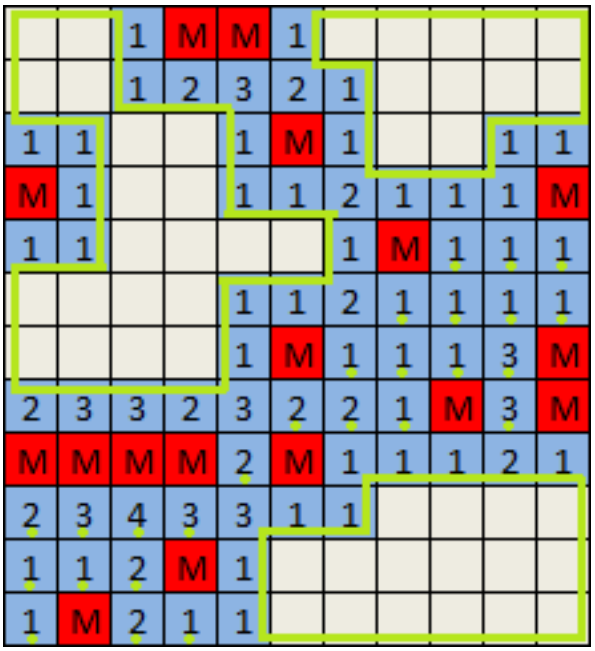

Figure 29: Graphic representation of 3BV

The $3BV$ benchmark adds one to its counter for each safe square that doesn't have 0 square as a neighbour and one for each group of zeroes. None of the zero squares in one group of zeroes can be a neighbour to any zero square that is part of a different group. In figure 29, each group is surrounded by continuous green line. When calculating the 3BV for this board, we just have to count every green object, which means add one for every green dot and also add one for every green area.

Figure 30: Maximal possible 3BV for a board

The highest possible 3BV score adds 1 point for 8 of every 9 squares. Such a result can only be achieved on the boards whose n and m values are divisible by 3 as is demonstrated in figure 30.
The $3BV$ is very good at describing the mechanical difficulty of a Minesweeper game. It doesn't take into account the computational difficulty of the steps it counts, but it is very good at predicting the runtime of algorithms with polynomial or even linear com-

plexities. It is possible to compute the 3BV value for a board with linear complexity.
$Rank$ is the maximal necessary amount of variables on which we have to apply our assumptions when making one step during a single game. Similarly to 3BV, rank is a value that is used to describe how difficult it is to solve a particular Minesweeper board. In a contrast to the 3BV, rank doesn't consider the number of steps needed, but only complexity of the most difficult step in the whole solving process. Its use is very limited when used to evaluate polynomial algorithms. It can be used to predict whether such an algorithm even has a chance of succeeding, but acquiring this value is usually to costly to be used just for this purpose. It however does well to highlight the runtime of a complete algorithm that can solve any solvable board. It can also be used as a part of optimizing process and to analyse a run of a CSP based algorithm.
The big drawback is its computational complexity. To determine this number, it is necessary to run an algorithm that can solve the board using domain reduction. For this reason, it is best to generate it when such an algorithm is already being used.
For a human player, it can outline the amount of reasoning that has to be done in order to solve the board.

\	0	1	2	3	4	5	6
0	0	1	M	1	0	0	0
1	0	1	2	2	1	0	0
2	0	1	2	M	1	0	0
3	0	1	M	2	1	0	0
4	1	2	1	1	0	0	0
5	M	1	0	0	0	0	0
6	1	1	0	0	0	0	0
7	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0

Figure 31: The cheapest available step is rank 1

In figure 31 and 32, the light grey tiles are revealed and all blue are still hidden to player. Figure 31 highlights a possible step that can be completed at rank 1. When we make an assumption that [2,2] is a mine, we can use the constant at [1,1] and deduce that [2,0] and [2,1] are safe. This is however inconsistent with the constant at [1,0] that had to be one neighbouring mine, which means that our original assumption must have been wrong, allowing us to make rank 1 move.

\	0	1	2	3	4	5	6
0	0	1	M	1	0	0	0
1	0	1	2	2	1	0	0
2	0	1	2	M	1	0	0
3	0	1	M	2	1	0	0
4	1	2	1	1	0	0	0
5	M	1	0	0	0	0	0
6	1	1	0	0	0	0	0
7	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0

Figure 32: The cheapest available step is rank 2

In figure 32, only rank 2 moves are possible. This means we have to do pick two variables. One pair that would yield usable results is [2,5] and [2,3]. If we assume that both are unsafe, we can, in similar fashion to the last example, show inconsistency with the constant at [0,4]. But such a result still doesn't allow as to make a move, we have only learned that both [2,5] and [2,3] cannot be unsafe. We can try another assumption, saying that [2,5] is unsafe and [2,3] is safe, this will conflict with constant at [0,4] again, leaving us with only assumptions where [2,5] is safe, giving us enough information to make a rank 2 move.
This increase in difficulty from rank 1 to rank 2 highlights, why the rank of the board is so important. The increase is exponential and higher rank steps are thus very hard to detect.
Some very easy boards can have a rank of 0, this means that simply be made just by comparing the constants and the number of known mines and unknown squares that surround them.

5.3 Complexity of Minesweeper game

Playing a safe Minesweeper, meaning doing a safe move until the game is won or proving that no such move is possible is a co-NP-complete problem[17]. The proof that Minesweeper is NP-complete by Kaye only holds where we also check that the game engine has made no mistake, but that is not required. An algorithm that also checks the consistency of the information provided by the game engine is able to play the game. Whether these two problems are equal and thus *NP = co-NP* is an open question in mathematics[18]. When we describe a Minesweeper board as a CSP and use a general solver to find all possible mine distribution, the complexity of the game is Sharp-P-complete[13]. This approach can be used to play a Minesweeper game as it will solve

every instance of solvable Minesweeper game and provide additional information for unsolvable boards that allow for more informed guesses, but that is not the goal of this paper. The theoretical complexity of generating a solvable grid is unknown. The goal of this paper was to implement a generator that builds a Minesweeper grid gradually and relies on an algorithm for solving the game when solvable. Since such an algorithm is co-NP-complete, the board generator must be at least as difficult as such a problem.

6 Minesweeper solver & board generator algorithms

This section aims to describe the inner workings of the Minesweeper solver, as well as the board generator that relies on it.
The sections that refer to either the solver or both solver and generator are presented first. Apart from this, the sections are sorted chronologically. If the data reaches the part of the program first, the section describing it is also listed first. In cases when some part is used multiple times during one resolution loop (i.e. Identifying squares, SPS), the order is based on the point when the data has entered for the first time.
Apart from describing the algorithms, which are also presented in pseudo-code, I also describe concepts too specific to be listed in the section *State of the art* (see 3) and introduce ideas behind various formats I have used to store and represent the data provided.

6.1 Minesweeper solver

My Minesweeper solver uses multiple algorithms with increasing success rates and computation complexities. The most basic layer consists of modified single point solver that was already introduced in the previous chapter (see 3.1). Consecutive layers consist of custom rules and have polynomial complexities. The last and most robust layer of the solver is based on a constraint satisfaction problem, but the computational cost is exponential with respect to the number of squares.

6.1.1 Human solver and constraint building

The way a human plays a Minesweeper game resembles the CSP very closely. We have our variables, the unknown squares. There are also two domains, *has mine* and *doesn't have mine* that we can assign. The constraints also seem to be easily constructible.

Figure 33: Easy Minesweeper situation to illustrate constraint parallel in Minesweeper

For example, if we set C_1 and C_3 equal to 1 and C_2 equal to 2, we can certainly construct some constraints. C_1 would give us:

$$((X_1, X_2), [(\textbf{mine}, not - mine), (not - mine, \textbf{mine})])$$

The constraint resulting from C_3 would be very similar, thanks to the reflection symmetry on the main diagonal. Writing C_2 as a constraint is more difficult:

$$(X_1, X_2, X_3, X_4, X_5),$$

$$[(\textbf{mine}, \textbf{mine}, not - mine, not - mine, not - mine),$$

$$(\textbf{mine}, not - mine, \textbf{mine}, not - mine, not - mine),$$

$$(\textbf{mine}, not - mine, not - mine, \textbf{mine}, not - mine),$$

$$(\textbf{mine}, not - mine, not - mine, not - mine, \textbf{mine}),$$

$$(not - mine, \textbf{mine}, \textbf{mine}, not - mine, not - mine),$$

$$(not - mine, \textbf{mine}, not - mine, \textbf{mine}, not - mine),$$

$$(not - mine, \textbf{mine}, not - mine, not - mine, \textbf{mine}),$$

$$(not - mine, not - mine, \textbf{mine}, \textbf{mine}, not - mine),$$

$$(not - mine, not - mine, \textbf{mine}, not - mine, \textbf{mine}),$$

$$(not - mine, not - mine, not - mine, \textbf{mine}, \textbf{mine})],$$

As we can see, the number of elements in set has increased by a lot. This can be expected, since what we are doing is listing all the k-combinations on set that is the size n. In our example, the set consists of all the mines that are neighbour to the number we are concerned about, in the case of C_2 it is $X_1, X_2, ..., X_5$. The k value can be either the number of undiscovered mines neighbouring to the C_2 or number of clear squares, the resulting amount of combination will be the same regardless. The number of all k-combination of a set of a certain size can be determined by an binomial coefficient, in our case it's $\binom{5}{3} = \binom{5}{3} = 10$. The most complicated constraint we can get is a known square surrounded by eight undiscovered squares where half of them have a mine, in that case we have $\binom{8}{4} = 70$ different states that would satisfy the constraint.
Because of this complexity, I have decided to try the cheapest and least costly strategies first, before resorting to the full arc consistent CSP.

6.1.2 Abstraction of the solver algorithm

The whole algorithm is very linear. It takes a minefield and uses resolution methods on it, one by one, often changing it in a way that allows the next method in line to work.

```
initialization;
while board not done do
    reveal open spaces;
    if a step is made, save and break;
    identify active squares();
    if a step is made, save and break;
    addNodes();
    if a step is made, save and break;
    form the clusters;
    run SPS() on each cluster;
    if a step is made, save and break;
    add constraints;
    for each Rank do
        find all combination of that rank using Combinations class;
        remove combination of bound squares;
        examine all their permutations using PermWRepet class;
        if a step is made, save and break;
    end
    if a step is made, save and break;
    check for known patterns;
    if a step is made, save and break;
    if he minecount only gives information on relevant squares then
        remove contradictions;
        check possible mine allocations;
        if a step is made, save and break;
    else
        remove contradicting permutations;
        if remaining permutation have the same amount of mines as minecount
        then
            clear inner squares;
            save and break;
        else
            mark board unsolvable;
        end
    end
    mark board unsolvable;
end
```

Algorithm 2: Abstraction of the solver algorithm

6.1.3 Storing the information about the current state of the board

An algorithm that is compatible with PGMS[7] can access all available information on the current state of a board through a function that returns the state of every square. I didn't find this accessibility ideal for my algorithm, so I decided to use an object oriented approach and express all the squares relevant for the next step as a two classes, *Constraints* and *Nodes*.

6.1.3.1 Structure of a constraint square

Each instance of the *constraint* class contains a position of a known square, a list of unknown squares that are also neighbours to the known square, in the form of object called a node, and the number of nodes from the list that contain a mine.
The contents of this list of nodes act as pointers, same as the corresponding list each node holds. Thus each constraint provides access to all of its nodes and each node provides the access to all of its constraints.
Furthermore, the class *Constraints* implements the Java interface Comparable, where two constraints are considered equal when they refer to the same known square and a constraint with fewer nodes tied to it is considered to be smaller.

6.1.3.2 Structure of an unknown constrained square - a node

The *node* class has a structure similar to that of a *constraints* class. It is also closely tied to the position of one square, in this case an unknown square. It stores it's coordinates, it also has a two binary variables, one labelling the node as safe and the other saying it holds a mine. This allows a node not only to be marked as safe or unsafe, but to also have both or neither of these attributes. Very much like the *constraints* class, it also holds a list of pointers, this time it holds all the *constraints* that have the node in their list, the functionality is similar.
The big difference when compared to the *constraints* class is the blacklist. This list contains pointers to all the nodes in the same block, this means they are tied together and knowing value of one of the nodes means knowing the value of all the nodes on the blacklist. When utilized, this list can reduce multiple variables into one.

6.1.4 Revealing open spaces

This process is handled automatically by the interface in most applications that implement Minesweeper as a game and because of that, I have decided to implement this as a function of my game engine that can be called by the player, in this case, a solver algorithm.
This algorithm ensures that there are no zero variables that have unknown squares

around them.

```
initialization;
while board has changed & game not lost do
    for each square on the board do
        if square(X_{i,j}) == 0 then
            for each X_{i_a,j_a}, i_a, j_a ∈ {−1, 0, 1} do
                if X_{i_a,j_a} is not out of bounds then
                    probe X_{i_a,j_a};
                end
            end
        end
    end
end
```

Algorithm 3: resolving 0 squares

This process can be done in a more efficient way. When an unresolved 0 square is found, it is possible to backtrack on the part of an iteration through the board, which ensures that all the squares affected by the recent change are resolved for in their current state, rather than just iterating continuously until one iteration doesn't find any new squares to be probed, but this process takes very little of the overall computation time and the worst case scenario when the $O(n^2)$ can be observed is very rare.

6.1.5 Identifying the relevant squares

After the resolution based on zero squares, the solver is provided with the full board and all the available information. Most of the squares on this board can be safely disregarded, because they either don't provide any information on any unknown squares or they are unknown squares, but with no available information regarding them.
This process is much cheaper that even a CSP with no arc consistency and only simple forward checking, which is $O(n^2)$, since iterating through the minefield costs only

$O(n)$.

```
initialization;
for each square in MineMap do
    if square X_{i,j} == MARKED then
        square(X_{i-1; x+1,j-1; j+1}) = square(X_{i-1; x+1,j-1; j+1}) - 1;
        remove(X_{i,j});
    end
    if square(X_{i,j}) == 0 then
        if reveal neighbours == success then
            continue(main loop);
        end
        remove(X_{i,j});
    end
    if square(X_{i,j}) ¿ 0 then
        addConstraint(X_{i,j});
        addNewKnownSquaresNeighbouring(X_{i,j});
    end
    if square X_{i,j} == UNPROBED && NodeList.contains(X_{i,j}) == false then
        square X_{i,j} = unconstrained variable;
    end
end
```

Algorithm 4: Identifying the relevant squares

When this algorithm finishes, all the known mines are deleted and the information on the known squares is adapted to correspond with this change. This mean that all the regions that have already been solved are now open areas bordered by deleted squares and variables.

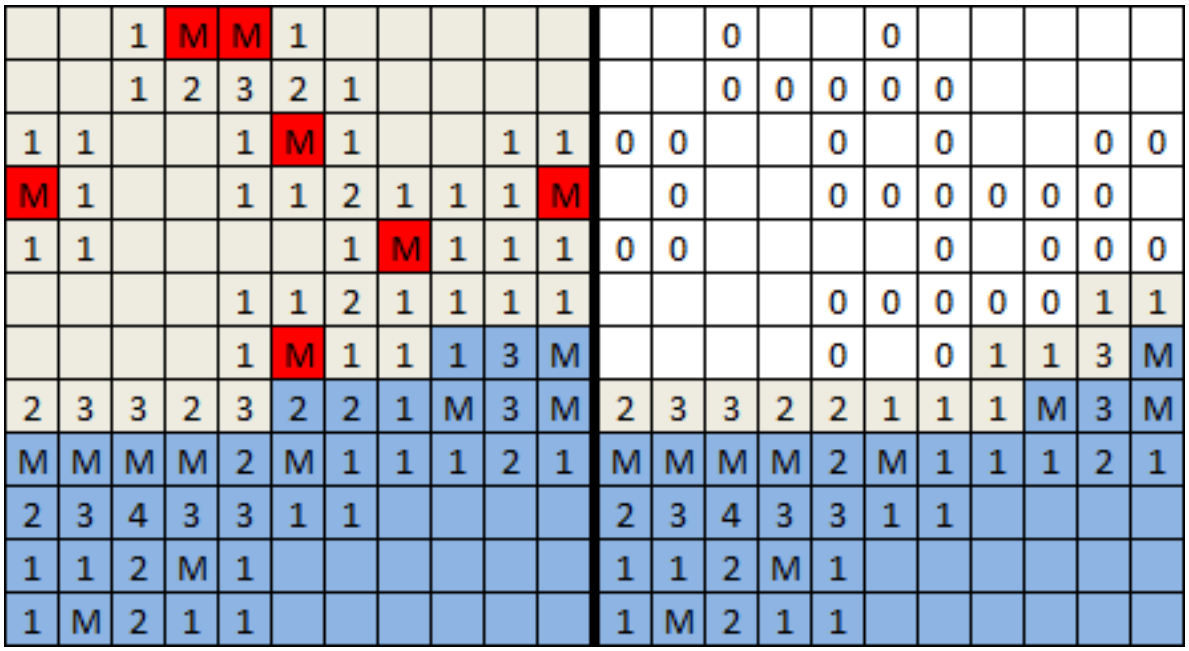

Figure 34: This is how a board would get adapted

This algorithm needs to be used to enable the more advanced techniques, but is also able to perform some resolution steps without increasing the complexity, staying at O(n) with or without searching for viable steps.

6.1.6 Identifying the variables - nodes

Since we have already went through identifying relevant squares while searching for the constraints, the search for nodes can use this information.

```
initialization;
for each Constraint C_{i,j} in Constraint list do
    for each node N_{i',j'} in neighbourhood of C_{i,j} do
        if N_{i',j'} == UNPROBED then
            if N_{i',j'} is unknown then
                add(N_{i',j'});
            end
            if N_{i',j'} !knows C_{i,j} then
                add(C_{i,j} to N_{i',j'} constraint list);
            end
            if C_{i,j} !knows N_{i',j'} then
                add(N_{i',j'} to C_{i,j} node list);
            end
        end
    end
end
```

Algorithm 5: adding the nodes

This process was the last step needed to make all the data, needed to determine the next move, represented in a way that is described in section 6.1.3. The *nodes* lack the blacklist at this point, but that is only needed for optimisation and will be added much later in the resolution process.

6.1.7 Forming the clusters

A **cluster** is a group of *nodes* and *constraints*. If a constraint is a part of a cluster, all its nodes are also part of that cluster, similarly, when a node is part of a cluster, all its constraint are also part of that cluster. At the same time, all the nodes and constraints are part of exactly one cluster, which means that when a constraint is part of a cluster, none of its nodes can be a part of any other cluster and the same goes with any node and its constraints.

Clusters help cut down the number of variables I have to consider at the same time. This is advantageous in any algorithm where the complexity is higher than linear. My algorithm has complexity $O(k^n), k > 1$ and as the size of n increases, the $2k^(n/2), k > 1$ becomes much smaller than $k^(n), k > 1$. Naturally, this is just an example, the actual clusters don't always have the same size. Unless the board is solved or uninitiated, there will always be at least one cluster and the maximal possible number will always be lower than the number of all squares.

In practice, two clusters are two groups of squares where changing the value of any unknown square in one cluster has no immediate effect on the other cluster.

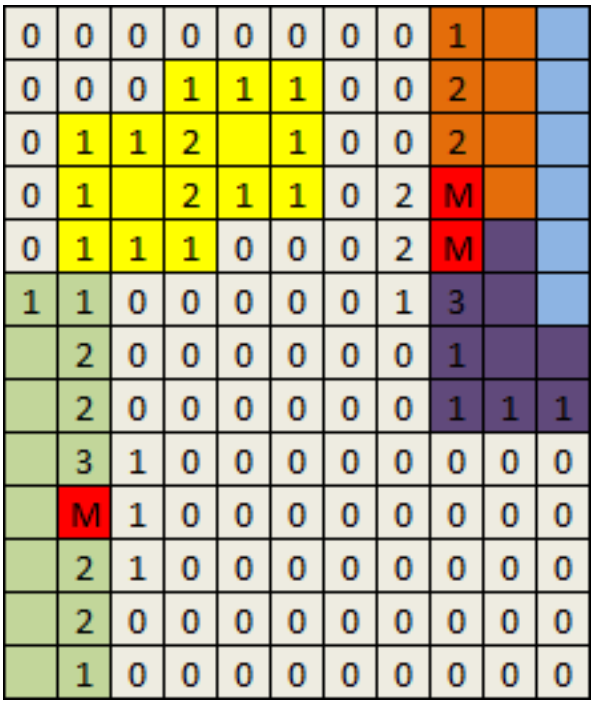

Figure 35: Four clusters on one board

In figure 35, there are two main areas, the top right and from top middle to the bottom left. They both contain two clusters, the brown and violet clusters on the top right are touching, but only their nodes are in each other's neighbourhood and nodes only register constraints in their vicinity. And there is a good reason for that. If the uppermost violet node changes its set of possible domains, it would change nothing for the brown cluster, none of its constraints touches the node and the only part of the brown cluster that is close enough is an unknown square. Of course, if we uncover this square in the future, the clusters might connect, but that would require a move to be made and new cluster layout would have to be made. The blue squares on the very top right are not part of any cluster, since they are unknown and we have no information on them, there is no reason to even consider them unless we are counting the hidden mines.

At the same time, the green cluster at the bottom left seems it could be made into two as well, but the square on the left of the known mine is connected to constraints on both upper and lower part and assigning a value to it would affect both. On the other hand, the yellow cluster is touching the green one, but only constraints are in vicinity. Constraints don't tell us anything about other constraints, so they can be dealt with separately.

From figure 35, we can see how much can the magnitude of the problem decrease with the use of clusters. Instead of dealing with the whole board or perhaps dividing it in

a simpler manner, for example splitting parts divided by empty spaces, we would get much bigger problems that would require more computation time.
Since all the nodes we are interested in have to be part of an constraints, we would only access the constraints directly and use them for defining clusters, any nodes that are part of that cluster. Then I use this cluster based list to make a same one, but node based, which is possible thanks to similarities nodes and clusters share, as described in section 6.1.3.

```
initialization;
while Constraint without cluster exists do
    place first unclustered constraint into new cluster;
    while current cluster is growing do
        for each node of a new constraint in cluster do
            add nodes constraint to the cluster;
            remove any duplicate constraints;
        end
    end
end
sort clusters by size;
make a tally of unclustered constraints;
```

Algorithm 6: forming the clusters

Now that we have a list of clusters based on constraints, we can define those clusters by their nodes too, for the ease of access.

```
initialization;
for each cluster do
    for each constraint in current cluster do
        add nodes on the constraint's list to the node cluster;
    end
    remove duplicate nodes in the cluster;
    sort node clusters by size;
end
```

Algorithm 7: defining clusters with nodes

It's very important to realize that the size of a cluster based on nodes and a cluster based on nodes can be different and so can be the order of these clusters.

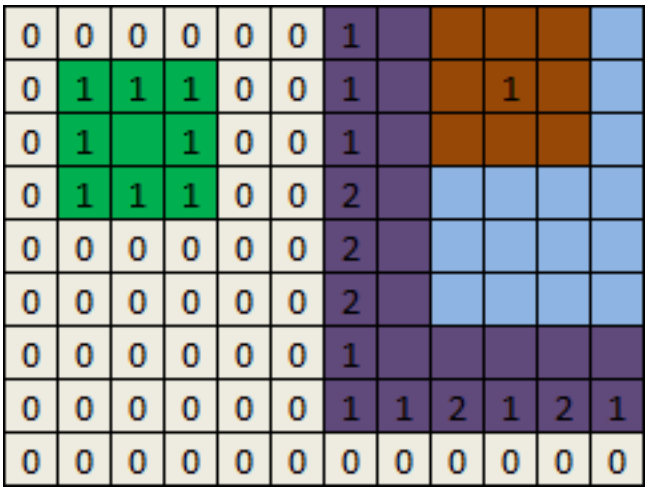

Figure 36: Different sizes of clusters

Figure 36 shows us an example of clusters with wildly different sizes when defined based on nodes or constraints. The green cluster has eight constraints and only one node, making it very similar to an overdetermined system that can be easily solved. If we describe clusters with constraints, it would only be the second smallest, but when use nodes instead, it is the smallest set on the boards.
On the contrary, the brown cluster only has one constraint but 8 nodes, this time similar to an underdetermined system, which could be solved in some cases, but it is unlikely, in the case of the brown cluster, the one constraint would either have to be a zero or an eight. If we used nodes to describe the cluster, the brown cluster would be the second smallest, but when using constraints it is the smallest cluster.
Regardless of the way in which we describe the clusters, the violet cluster is always the biggest. This cluster is a more conventional one and illustrates that in general a cluster with more nodes tend to have more constraints and vice versa, but the size of both variants and the difference between those sizes matters and can help us optimize. The violet cluster has a concave of constraints on nodes, which gives the constrains the superior numbers, meaning the violet cluster is overdetermined as well as the green cluster and both can be used to find the next move.

6.1.8 Applying the SPS

The SPS is identical to the one described in section (3.1), with the only difference being that it marks all the probed squares it enters. This allows it to identify relevant constraint squares for the next step of the algorithm resolution.

6.1.9 Analysing combinations of mines in a cluster

The amount of unique mine allocations one cluster can have is finite. I try to find most important squares using the techniques discussed in section (4) and test if all but one of mine allocations that are possible for this subset of squares are illegal. Such a thing proves that the remaining allocation is the correct one.

```
initialization;
for each Rank do
    for each cluster do
        form combinations of current rank;
        remove combinations of bound squares;
        for each combination do
            for each permutation in combination do
                add affected squares not in combination to bound list;
                check for contradiction;
            end
            if number of contradictions +1 = number of permutations then
                return;
            end
        end
    end
end
```

Algorithm 8: Analysing combinations of mines in a cluster

6.1.10 Useing the minecount to solve the Minesweeper board

The minecount has limited use, since as long as there are inner squares and minecount has more mines left than the permutations require, it doesn't provide only helps us approximate the likeliness of inner square having a mine. But if there is a permutation that requires more mines than what minecount allows, we can safely count that as a contradiction and strike that permutation out. At the same time, if all the permutation require the same amount of mines and this number is equal to the remaining unmarked mines, we can safely probe all inner mines.

```
initialization;
if max permutation mine > (minecount - marked mines) then
    remove contradictory permutations;
    analyse current combination;
end
if max permutation mine = min permutation mine = (minecount - marked mines)
then
    probe inner squares; return;
end
```

Algorithm 9: Useing the minecount to solve the Minesweeper board

6.2 Minesweeper board generator

The *Minesweeper board generator* works in two phases, where the second phase engages only when the first one fails.
The first phase places mines randomly and checks whether the board is still solvable using past results and the Minesweeper solver when they aren't available. When a new mine turns the board to unsolvable, the algorithm counts the average amount of mines that would have been placed on the unprobed part of the border of the unsolvable cluster if the random placement continued and tries to place that number of mines, one by one, checking if the cluster is solvable. If that doesn't happen, the original problematic square along with all of the safe unprobed squares of the border of the cluster are marked as safe and no additional mines will be randomly placed there. If the amount of mines reaches the desired amount, the algorithm ends.
If the amount of safe squares get so high that it would be impossible to put down enough mines to satisfy the input arguments, the algorithm switches to the second phase and starts placing mines on the unprobed border of the cluster until it either becomes solvable or the whole unprobed border becomes just mines, at that point, it fills the inner squares of the border with mines as well. If this doesn't exceed the desired amount of mines, the algorithm switches back to phase one, if it matches it, the algorithm ends. If the amount of mines is exceeded, the mines are removed from the border of the cluster one by one until they match the number of mines in the input argument.

```
initialization;
mark start as untouchable;
mark neighbours of start as untouchable;
while actMines < minecount do
    if random square has no mine and isn't safe then
        place mines;
    else
        continue;
    end
    if board unsolvable then
        calculate amount of boarder mines;
        for each border mine do
            place randomly on the border;
            if board solvable then
                continue;
            end
        end
        remove mines added in this iteration;
        mark border as safe;
    else
        continue;
    end
    if unassigned squares <= unassigned mines then
        mark safe squares as unassigned; while cluster unsolvable & border has
        unassigned squares do
            place mines on the border; if board solvable then
                continue;
            end
        end
        fill cluster with mines;
        if actMines <= minecount then
            continue;
        end
        while actMines >minecount do
            delete one mine in cluster's border;
        end
    else
        continue;
    end
end
```

Algorithm 10: Overview of the generator algorithm

6.2.1 Turning a board from unsolvable to solvable by the addition of extra mines

In the section 2.5.2, I have shown an area that cannot be safely solved. The board on figure 37 is almost identical to board seen in figure 18. The only difference is that instead of having either [5,0] or [5,1] being a mine, we have both.

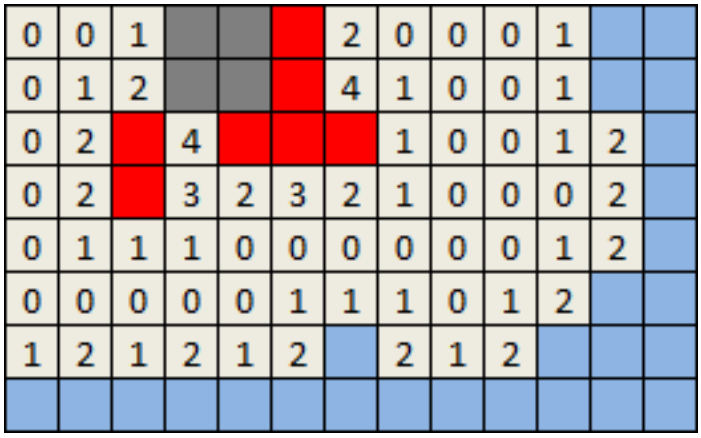

Figure 37: Potentially solvable board

It might seem that this change would make the board even more difficult to solve, turning a safe square that would reveal new information when probed into a mine, which is a square that can never be probed and will never provide any information. But in figures 20, 21 and 22, we can only describe the position of the mine behind the yellow squares by a probability distribution. From the players view, it has no concrete position and since both the yellow squares will always have the same probability, because the constrains on them are identical, the only two distributions that will allow us to resolve these squares are 0mine is exactly what we need to achieve the latter.
But even after resolving the squares [5,0] and [5,1], the grey squares in figure 37 can still be unsolvable that is in case they hold two mines

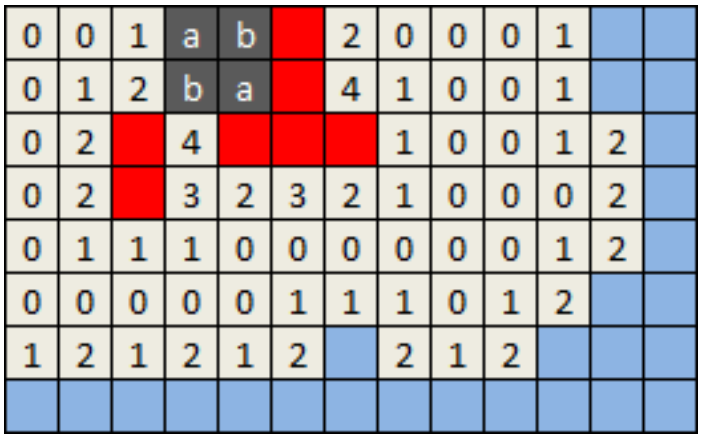

Figure 38: Probability distribution inconclusive

This situation is reflected in figure 38, where we are in a very same situation we were in with the yellow squares in figures 20, 21 and 22. And just like then, we can resolve it again by placing another mine (or removing one) and we will get a situation very similar to figure 22, but this time the whole board is solvable.
In fact, we can use this approach every time when we want to make a cluster solvable. It is possible that only a small amount of mines will be needed, but in some cases the whole cluster needs to be filled with mines.

6.2.1.1 Searching a boarder of a cluster and making a point of entry

A border of a cluster consists of all the active constrains based on known squares and nodes they refer to in that cluster. With normal constraints, all the squares in a cluster would be on the border, but when we use the count of mines on the board as another constraints, it is possible to have unknown squares that belong to a cluster but aren't on its border. Since we are not guessing, we cannot start in more than one place and that means we are attempting to solve every cluster from outside progressing to the inside with a continuous undivided border, except for the scenarios where mine count is used, which is discussed in section 6.1.10 . For the border to be divided, we'd have to either probe a square inside the cluster, which would be an uninformed move since there wouldn't be any uncovered square in the neighbourhood, or we'd have to actually divide it, which would mean they are two clusters and not one. In the case where mine count is used, it is possible to uncover an inside square if we can prove that all the inside squares hide no mine underneath.
If we want to solve a cluster, we have to find an unknown square on its border that can be probed, thus allowing us the access the inside squares. If no such point exists, the square is unsolvable.

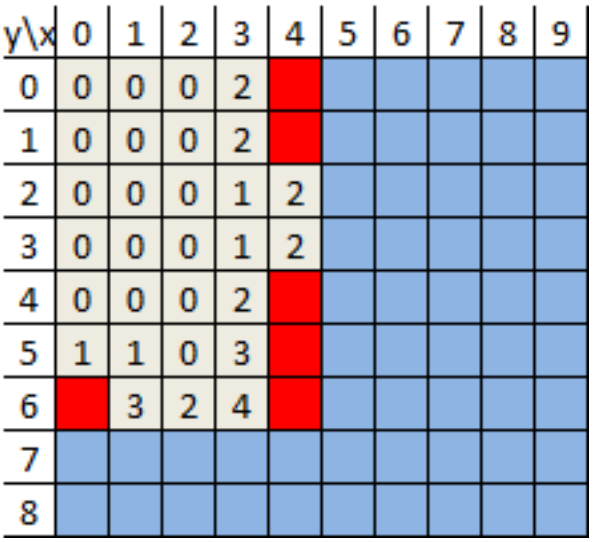

Figure 39: Unsolvable clusters with no entry point

The figure 39 is unsolvable. There are two clusters (see 6.1.7) and neither can be used to determine a safe move.
Understanding why we cannot find a move helps us a lot if we want to change it. The upper cluster is much simpler, all the squares in question can have any value, we the whole cluster is symmetrical on the y axes between 2 and 3. If the square [5,3] holds a mine, the other nodes (see 6.1.3) are safe, because of the symmetry, square [5,2] behaves in a similar fashion. The single constrain squares can also hold a mine, if [5,1] holds a mine, [5,4] would also hold a mine and [5,3] with [5,2] would be safe. Adding additional mine to the squares [5,3] and [5,2] would be of no help, known squares [4,2] and [4,3] would just increase to three, symmetry would stay in place and there would still be no step to be discovered.
Adding mines to one of the one constraint squares yields much better results.

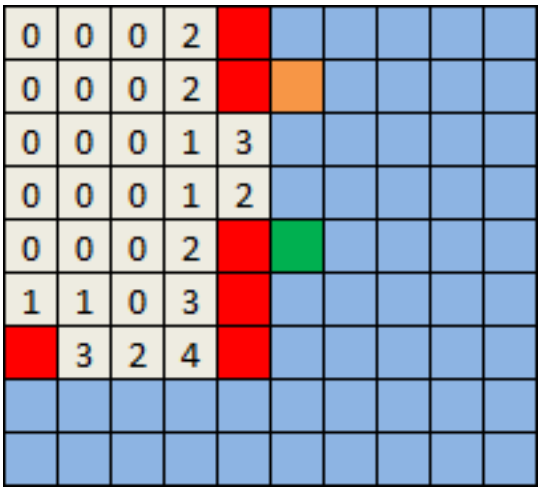

Figure 40: Green square is safe, orange must hold a mine

Regardless of which of the squares we chose, we will be able to resolve both, and while the middle unknown squares are still irresolvable, we have a chance that the green square will give use sufficient information upon being probed.
The bottom cluster of figure 39 is more complicated.

y\x	0	1	2	3	4	5	6	7	8	9
0	0	0	0	2						
1	0	0	0	2						
2	0	0	0	1	2					
3	0	0	0	1	2					
4	0	0	0	2						
5	1	1	0	3						
6		3	2	4						
7	a,b	a,c	b,c	a,b	a,c					
8										

Figure 41: Some of the possible solutions

We need to present at least three mine distributions for the bottom cluster to show

it is unsolvable. 2 of these distributions aren't symmetrical, but as seen in figure 41, where each letter means that that square holds a mine in the corresponding distribution, when we combine all three together, a pattern can be observed. Also, since each square holds mine in some distribution but doesn't hold it in every single one, there is no safe step to be made.
In this case, any position except for [2,7], would allow us to make a safe move and [2,7] doesn't work only because it would make all the nodes hold a mine, which would destroy the cluster completely.

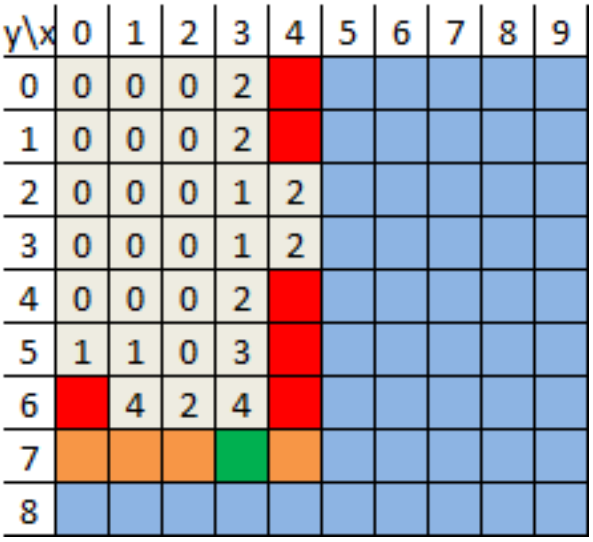

Figure 42: Cluster turned to solvable, orange squares must hold a mine, green squares are safe

Sometimes it is not possible to place any mine that would turn an unsolvable cluster into solvable one and the only thing we can do in order to keep the whole board solvable is to destroy the cluster completely, making all the unknown squares on the border hold a mine.

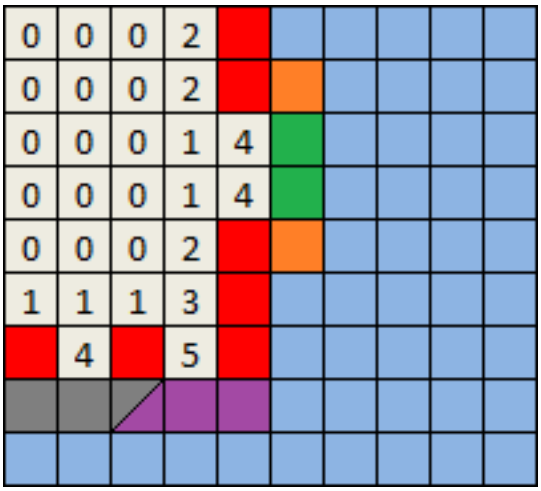

Figure 43: Adding mines wouldn't always allow us to reveal any new squares

As seen in figure 43, if both the orange square in the upper cluster holds a mine (which means one of the green squares holds a mine as well), adding more mines would never allow us to probe additional squares safely. Assigning a mine to the remaining green square would only close off that cluster and the bottom clusters, no matter where the current mines are and where we add new ones, the clusters still won't reveal a new square, since they both share at most one mine and each have at least one mine that is only in their cluster and none other.
The only way to make this whole board solvable by adding mines is to add them to all the unknown squares. That way we can just use mine-count to verify that we have already probed every safe square.
In the case we are generating a board based on initial parameters, including mine-count, it is possible that that number has been exceeded. For that reason, it is important to have a way to decrease the number of mines in such cluster.

6.2.2 Decreasing the number of mines in a cluster filled with mines while retaining solvability

Almost any cluster where all its unknown squares, as well as all the squares inside it, hold a mine, can be reduced in size, thus reducing the overall mine count.
We can separate these clusters into two different groups, where clusters in each group can lose squares in mines in similar fashion. The cluster is in the approachable group, if it touches one or two adjacent walls of the board and in the unapproachable group if it touches 2 opposing, 3 or 4 walls. Every cluster in the approachable group is reducible and most of those in unapproachable group are reducible as well, but it also contains some clusters that can't be reduced any further.

6.2.2.1 Approachable clusters

If a cluster is in the approachable group, it means that it is surrounded by solved parts of the board from at least two sides. In the example in figure 44, it is on top and to the right.

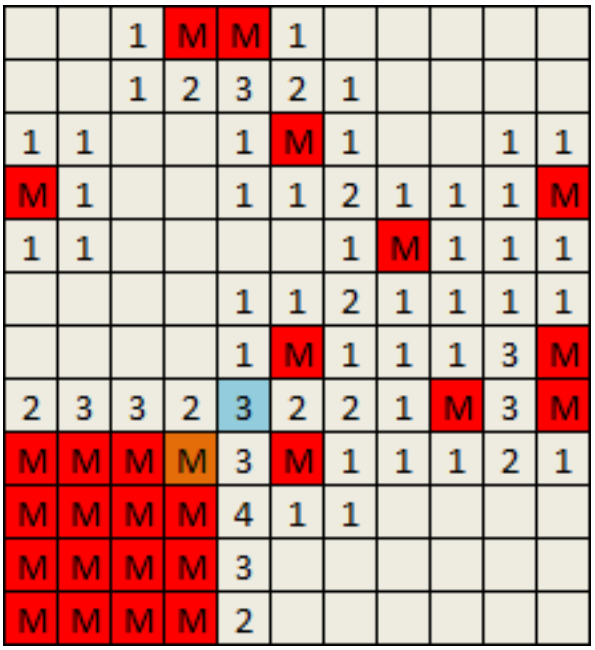

Figure 44: Solvable board with reducible cluster

The figure 44 shows very neat version of approachable cluster, since it's border can be found on just two perpendicular lines. The property that tells us that this cluster can certainly be reduced is the amount of known and unknown squares on the border. Since the border connects to the sides of the board that are perpendicular to each other, the unknown squares that are on the inside are in lesser number than the known squares. This is the same effect discussed in the section 6.1.7 and shown in figure 36 with the green and brown clusters.
In figure 44, the important interaction is between the blue known square and the orange unknown one. The blue square only has one active constraint that features only the orange square (except for the global constraint derived from the amount of unmarked mines). When we remove the mine hidden under the orange square, the blue one will be left with no undiscovered mines in its neighbourhood and will be able to show that the orange square can be probed.

Naturally, it is unlikely that the border of a cluster will lie on just two lines, but this only provides more concave areas for the known squares and more option where we

can delete a mines and quickly identify that square as safe.

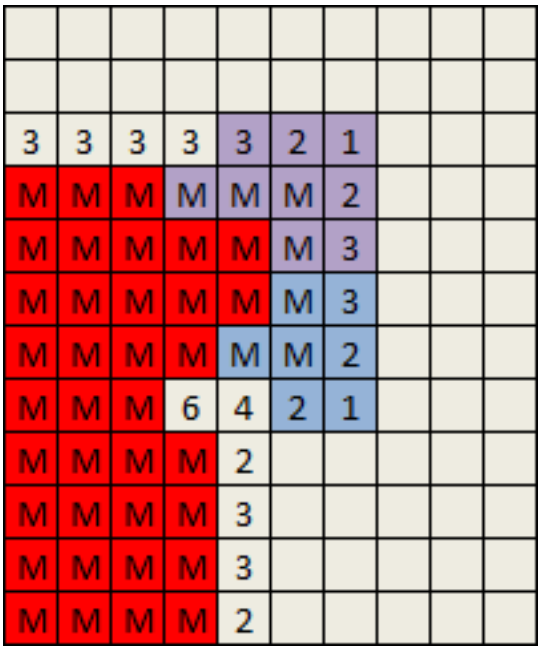

Figure 45: More complex approachable cluster

The figure 45 shows just that. The blue and violet squares signify the concave that probed squares have, which allows us to delete a mine in a fashion similar to one shown in figure 44. The area beneath the square with 6 neighbouring mines isn't marked as a concave, since the known squares surrounding it are touching on both top and bottom and actually create a reverse concave.

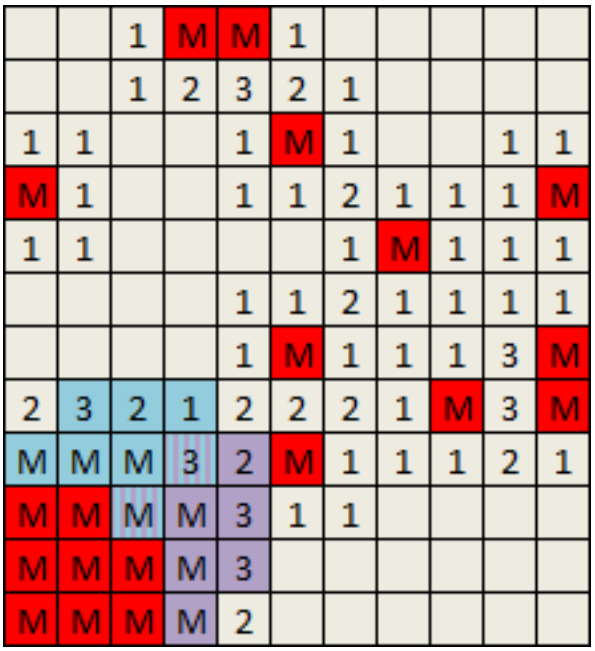

Figure 46: The result of reducing a cluster in 44

After identifying said localized concave and removing the mine, the now probed square, unless it touches the border of the board, becomes a part of another such concave. Because the removed line in figure 44 was placed on both the lines going through the border of this cluster, it has created two more concaves.
This easy reduction is provided by the fact that a known square that has only one unknown neighbour and 1 undiscovered mine in vicinity changes to zero undiscovered mines in vicinity, allowing us to identify the square that lost the mine as safe right away.

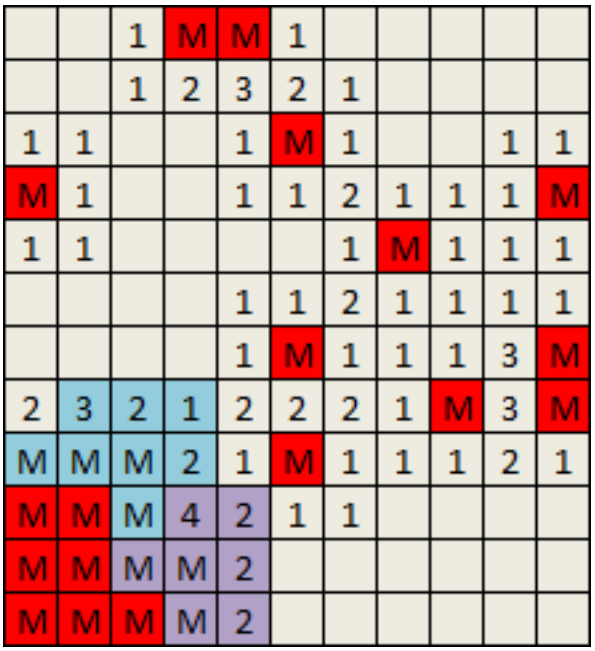

Figure 47: Result of removing mines in downward direction in 46

In figure 47, we continue in a similar fashion. In this case we just go through each column from a side that is not occupied by the cluster that is being reduced and proceeding towards the cluster. When we encounter the first column that has at least one unknown square with mines that belong to the cluster, we pick the uppermost or bottom one and delete it. As figure 48 shows, we can even remove a mine that is adjacent to the wall since one of the columns next to it is filled with known squares only and as a result we are left with a square at the bottom that only touches one unknown square, which allows us to use subset rule[7], if a line of mines in the column that is being reduced is broken, it might not even be necessary. The subset or subtraction rule can also be used when a part of a concave is blocked by a known mine.
Naturally, same can be done when searching by rows and since our cluster is in the approachable group, we know that it can't touch more than two adjacent borders of the board.

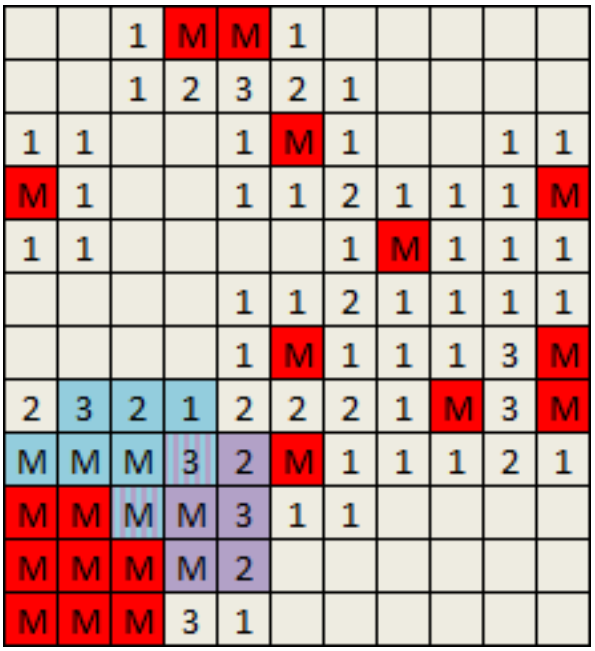

Figure 48: Result of removing mines in upward direction in 46

6.2.2.2 Unapproachable clusters

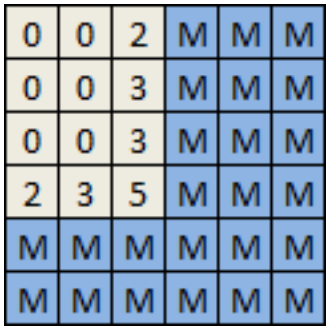

Figure 49: Typical unapproachable cluster

With unapproachable clusters, we can't always relay on finding a concave that would provide us with a mine that can be easily removed, although such instances can still occur.

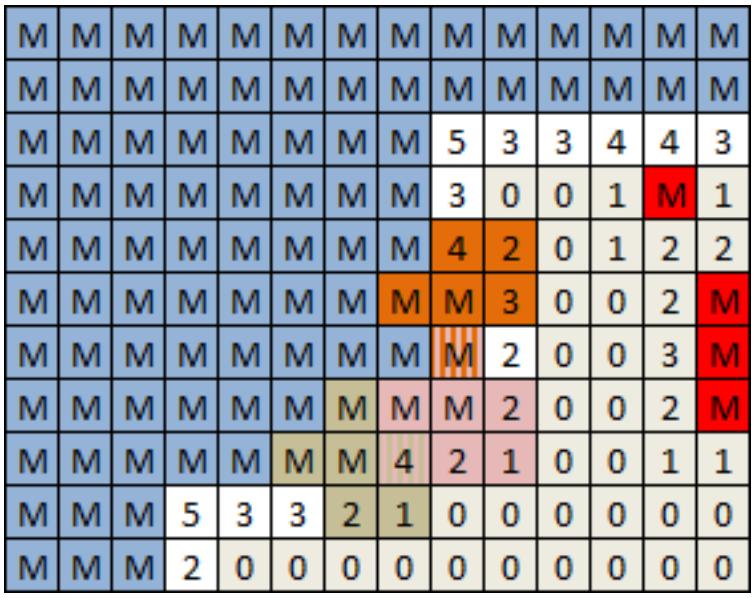

Figure 50: Unapproachable with concaves highlighted

And on this cluster, we can use the same basic methods shown in the section 6.2.2.1. Most randomly encountered unapproachable clusters will have some of these concaves, but since they often contain a lot of mines, it is probably that by filling them, we have gotten significantly over the desired number of mines the board was supposed to have and this means we need to delete a lot. If we keep deleting just the ones that are in a concave, we will eventually run out, since we are once again just taking a layers and we will eventually just smooth out the pattern until the squares that don't belong to the cluster that is being reduced are in a convex shape.

M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	M	M	M	M	M	M	M	M	M	M	M
M	M	M	5	3	3	3	3	3	3	3	4	4	3
M	M	M	3	0	0	0	0	0	0	0	1	M	1
M	M	M	3	0	0	0	0	0	0	0	1	2	2
M	M	M	3	0	0	0	0	0	0	0	0	2	M
M	M	M	3	0	0	0	0	0	0	0	0	3	M
M	M	M	3	0	0	0	0	0	0	0	0	2	M
M	M	M	3	0	0	0	0	0	0	0	0	1	1
M	M	M	3	0	0	0	0	0	0	0	0	0	0
M	M	M	2	0	0	0	0	0	0	0	0	0	0

Figure 51: Cluster 50, after concaves have been used up

Even without the concave, we can still reduce clusters like this.

0	0	1	M	M	M
0	0	2	5	M	M
0	0	2	M	M	M
2	3	5	M	M	M
M	M	M	M	M	M
M	M	M	M	M	M

Figure 52: Reduced a mine with mines on both sides

We can remove a mine that is adjacent to a known square if all the other mines adjacent to this square are also adjacent to at least one known square that has only unknown adjacent squares with a mine.
This process was implemented on figure 53, it only displays on segment of a board and doesn't show any wall of the board. After introducing this part in the first cut-out, we continue by removing a mine in the top right (marked green) a value of a known square was changed from 3 to 2 (marked yellow). If we use the subtraction rule[7], we can use

the 5 to the left (marked orange) and subtract 2 of the unknown squares linked to the yellow squares leaving only 0 pointing at the green square that had the mine removed, marking it as safe.
In the next cut-out, we continue this process, 4 is changed to 3 (yellow), the 3 below (orange) can be used to subtract from the 3, again leaving 0 pointing at the green square, allowing us to safely probe it.

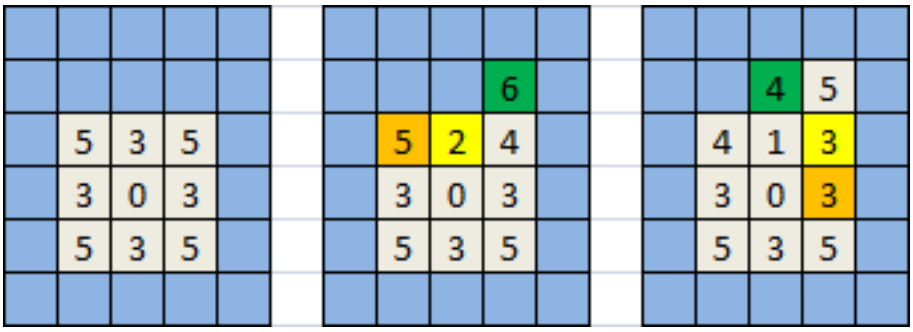

Figure 53: Process of removing mines in unapproachable cluster

We continue this example in figure 54. By removing the mine bellow the green square, we create a new 0 square (marked orange), allowing us to mark the green square as safe with only basic rules. This is possible, because we have created a local concave. This move also finishes the current line and we can start a new one. In second cut-out of figure we remove another mine (green square), this changes the yellow 3 to a two and we use the orange three for a subtraction. The last step shown is revealing the 1 in the last green square. Removing that mine has changed the yellow 3 to 2 and we use the orange 5 below for subtraction.

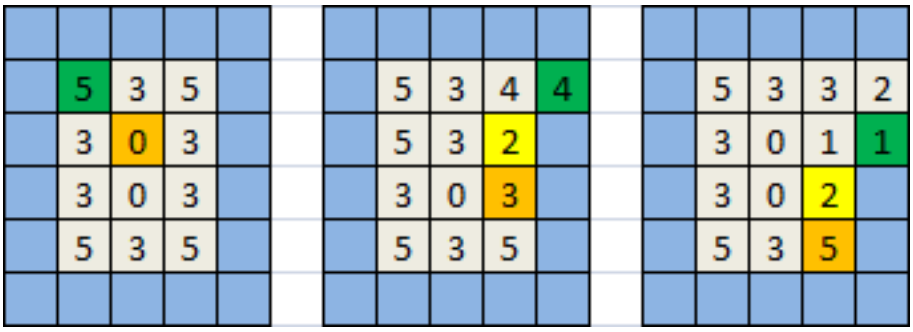

Figure 54: Finishing one row and starting another

But if we just keep searching for concaves or squares that share all but one unknown square with a fully saturated square, we can run out and get stuck, with no additional

mine being removable, this is why we have been implementing another strategy in these examples. When we are removing a mine, we make it in a way that the border of the new cluster is as small as possible. We had various options to chose from in first step of figure 53 and step 2 of figure 54, but with each of the remaining steps, we had only one option, as removing any other mine would have resulted in longer border. When we keep the border the smallest possible, we assure that we are always removing mines towards areas with uninterrupted areas of mines and thus saturated revealed squares. This is given by the fact that we have filled these clusters with mines before the removal of mines has started.

6.2.2.3 Irreducible clusters

The cluster shown in figure 53, is the smallest cluster, where no revealed square is touching a wall of the board that can be reduced.

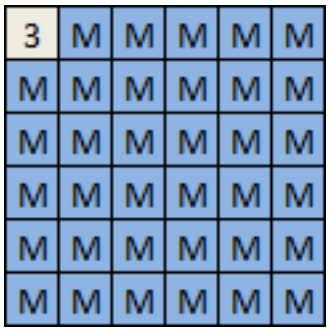

Figure 55: Irreducible cluster

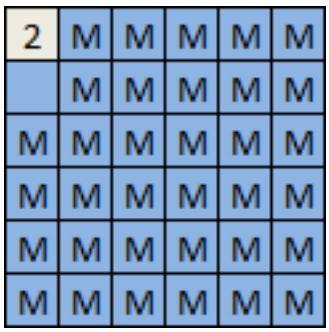

Figure 56: The board turned from solvable to unsolvable

7 Implementation

Both my Minesweeper solver algorithm and board generator are implemented in Java and tested in:

```
Java 1.8.0_73, VM 25.73-b02.
```

The solver is based on the PGMS[7] interface, but it will not compile when used in this manner, since I've modified it to provide additional interaction with the engine. The solver algorithm is entirely my creation and doesn't include any code from other PGMS project or any other sources.
The Minesweeper board generator is based on the interface for PGMS game engine and is able to run any solver that is compatible with the PGMS project. The names of each class have remained the same. When relevant files and PGMS engine are replaced with my files, granted there are no problems with pathing, all the original functionality remains intact. Except for the class names, public method names, constants and some of the variable names, all the code in my implementation of PGMS code is my creation and doesn't use any code from other sources.
In compliance with PGMS documentation, the main components of the solvers can be found in the class *CSP_FCH_MRV* that implements PGMS's Strategy interface and contains a method *play(Map)* that contains the main loop of the whole algorithm, along with the *SPS* method that can perform SPS on given board. My Minesweeper solver also uses *Constraints* and *Node* classes, *Pointerish* class that allows data exchange with the game engine and *PermWRepet* and *Combinations* classes that are integral in calculation of the strongest resolution method.
The class *MineMap* implements the *Map* interface from PGMS and is made to have all the functionality of *MineMap* class distributed with the PGMS, but to also be able to work with my Minesweeper solver and be able to generate a solvable boards of custom size, mine amount and starting position with the method *MineMap(int, int, int, int, int)*.

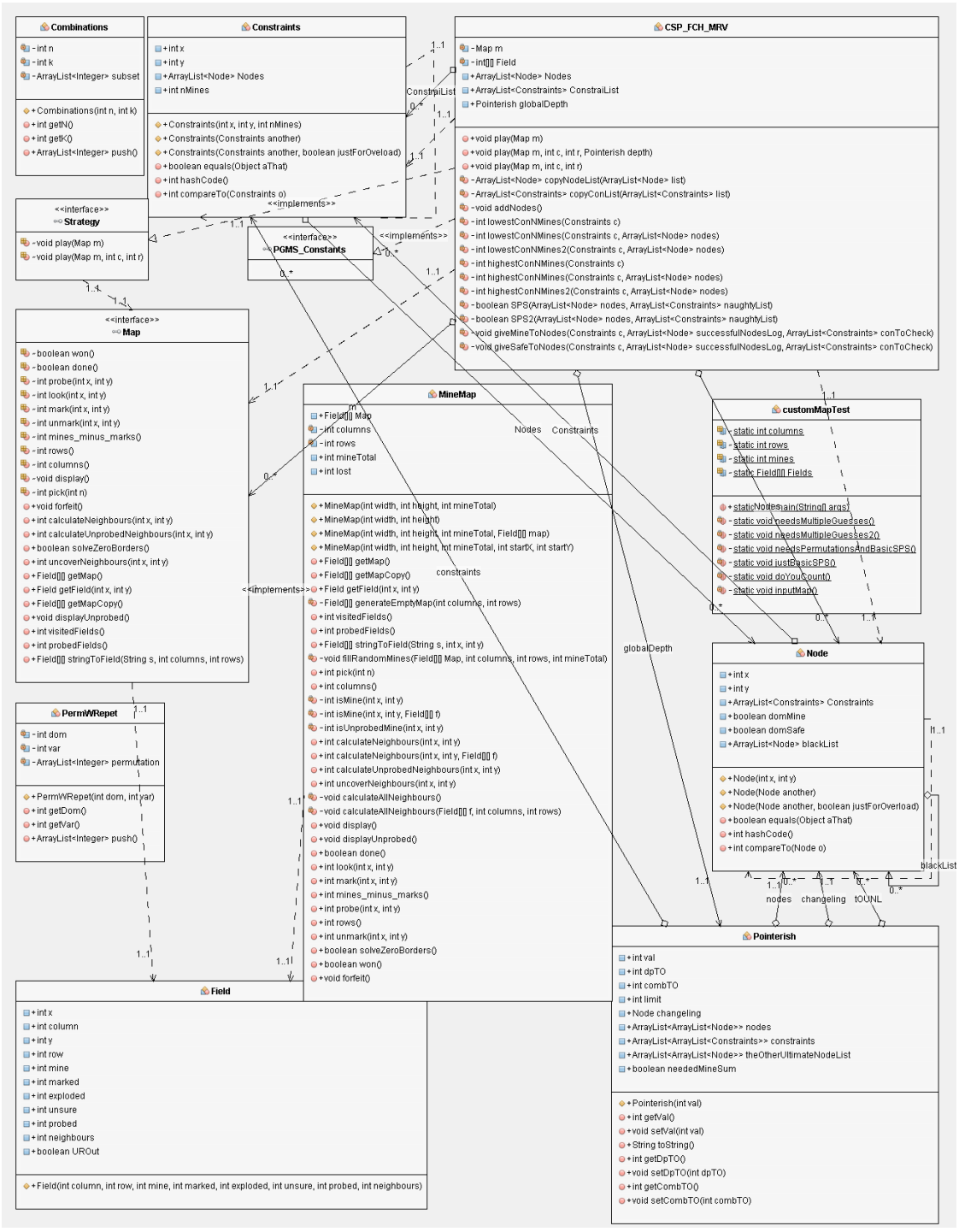

Figure 57: UML diagram of the whole project

8 Evaluation

In this section, I will evaluate the Minesweeper solver algorithm described in section 6.1 both by itself and in comparison to solutions from other authors and also the performance of my Minesweeper board generator on the standard board sizes.
All the testing of the solvers was done in the PGMS package with the standard PGMS application[7] and the MineMap.java and Field.java files replaced with my versions that provide compatibility for my Minesweeper solver and are available as one of the appendices.
All of the testing was done on CPU: Intel Core2 Quad Q9550 2.8GHz, GPU: ATI HD 3650, RAM: Corsair CM3X1024-1066C7 2GB (2 sticks), OS: Windows Vista Home Premium SP2.

8.1 Evaluation of the Minesweeper solver

My algorithm was designed to identify whether the board it gets as an input is solvable or not. As shown in the section (6), while each board it solves is solvable, it is not guaranteed to find every solvable board. To my best knowledge, there is no other algorithm with the same aim, so I've picked three other algorithms that are designed around playing both solvable and unsolvable Minesweeper boards while minimizing CPU time and maximizing win-rates. These algorithms will always either finish a game successfully or probe a mine and lose, while my algorithm either finishes a game or calls the board unsolvable. It will never probe a square that can't be proven to be safe and therefore never probe a mine.
In order to be able to use all four of these algorithms in one test, I've defined a a solved board as a Minesweeper board that was solved without guessing and success rate as a percentage of boards that were solved, where 100% result will mean that all of the boards the algorithm was presented with were solved without guessing and 0% will mean that none of the boards were. For each test, I've randomly generated a set of board of disclosed size, where each board was in compliance with the Minesweeper rule set described in section (2). For each of the tests, all the algorithms are working with the identical set of boards.

8.1.1 Success rates for identifying solvable squares

The ability to identify a solvable Minesweeper square is very important for my solver. The main goal is to present the player with an interesting board that doesn't require guessing. If we only present the player with simple boards, the game-play experience will change, since they will know that at every stage of the game, there is at least one small section that allows him to progress further without the need to consider the board as a whole.

success rates with advanced grid, randomly generated, 1000 runs

Algorithm	boards solved	success rate
Single point strategy	0	0%
Equation strategy	64	6.4%
CSPStrategy	71	7.1%
My CSP	76	7.6%

success rates with intermediate grid, randomly generated, 1000 runs

Algorithm	boards solved	success rate
Single point strategy	1	0.1%
Equation strategy	502	50.2%
CSPStrategy	502	50.2%
My CSP	637	63.7%

While the SPS shows excellent results when CPU time is concerned[6], its ability to recognize solvable boards is so bad I've decided against including it in any additional test, since it is completely unsuitable and getting a sample of successful results, especially for the *advanced* board would be difficult.
The other 3 algorithms have shown much better results. The Equation strategy and CSPStrategy have been able to identify the same number of boards, while my solver has identified 127% of their result on *intermediate* board. For the *advanced* board, the difference is much less significant. CSPStrategy has 111% successes of Equation strategy and my solver 107% of CSPStrategy.

8.1.2 Rank reached by my algorithm

To investigate why the success rate advantage of my solver is lover with advanced board, I looked into how much the rank and time constraints affect the run of the algorithm.

maximal and average rank of my CSP, 1000 runs

minefield	maximal rank (5.2)	average rank
9x9, 10 mines	2	1.02
16x16, 40 mines	9	1.8
30x16, 90 mines	9	1.24

The results for *beginner* are quite interesting, since the rank limit wasn't even used. It seems that such a small board doesn't allow for any more complicated situations, even though they can occur, the probability seems to be very low.

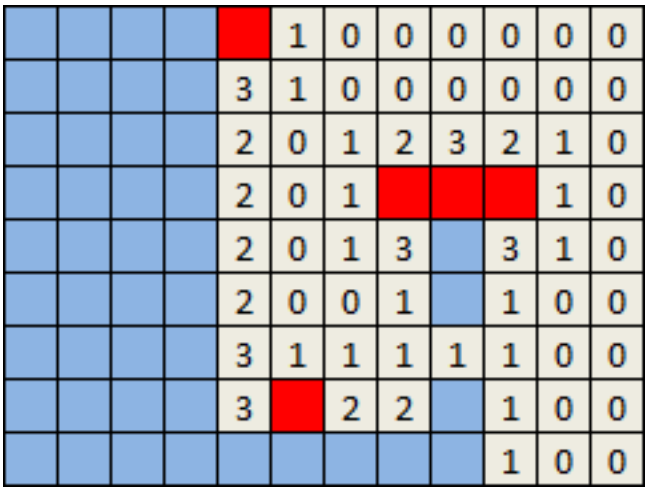

Figure 58: Complex board on a 9 by 12 board

Results for *advanced* and *intermediate* boards help with explaining the results shown in section 8.1.1. The algorithm has reached the maximal rank in both cases but the average rank of *advanced* board was only 69% of average rank for *intermediate* board. I assume that this decrease wasn't caused by easier boards, but by the fact that limit of 200ms per rank and 4.7ms for a selected group of mines doesn't allow the algorithm to find valid configurations on a certain rank for bigger clusters.

8.1.3 Examples of boards solvable only by some of the algorithms

The success rates shown in section 8.1.1 show that there should be some squares that can be solved by my solver and not the other ones. In this section, I show some of them in a form that can be used as an input for a square that can be then presented to each of these algorithms.
The numbers on figures 59 through 62 signify a probed square and the number of neighbouring mines. X means an marked and unprobed square that the solver has assumed holds a mine and * is for squares that have not been probed yet and aren't

marked.

```
0 | 1 | X | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
0 | 2 | 3 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
0 | 1 | X | 3 | * | 3 | 2 | 1 | * | * | * | * | * | * | * | * | * | * | * |
0 | 1 | 3 | X | * | * | 2 | * | * | * | * | * | * | * | * | * | * | * | * |
0 | 0 | 2 | X | 3 | 1 | 2 | X | * | * | * | * | * | * | * | * | * | * | * |
2 | 2 | 2 | 1 | 1 | 0 | 1 | 3 | * | * | * | * | * | * | * | * | * | * | * |
X | X | 3 | 1 | 1 | 0 | 1 | 3 | * | * | * | * | * | * | * | * | * | * | * |
3 | X | 3 | X | 1 | 0 | 2 | X | * | * | * | * | * | * | * | * | * | * | * |
1 | 2 | 4 | 3 | 2 | 0 | 3 | X | * | * | * | * | * | * | * | * | * | * | * |
0 | 1 | X | X | 2 | 1 | 3 | X | 4 | * | * | * | * | * | * | * | * | * | * |
0 | 1 | 3 | X | 2 | 1 | X | 3 | 3 | * | * | * | * | * | * | * | * | * | * |
0 | 0 | 1 | 1 | 1 | 1 | 1 | 2 | X | * | * | * | * | * | * | * | * | * | * |
1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 2 | * | * | * | * | * | * | * | * | * | * |
X | 2 | 1 | 1 | X | 2 | 2 | 1 | 2 | 2 | * | * | * | * | * | * | * | * | * |
3 | X | 2 | 2 | 3 | X | 2 | X | 2 | * | * | * | * | * | * | * | * | * | * |
X | 2 | 2 | X | 2 | 1 | 2 | 1 | 2 | * | * | * | * | * | * | * | * | * | * |
```

Figure 59: Results of running Equation strategy

```
0 | 1 | X | X | 2 | 2 | 1 | 1 | 0 | 1 | 3 | X | 3 | X | 1 | 0 | 0 | 0 |
0 | 2 | 3 | 4 | X | 2 | X | 1 | 0 | 2 | X | X | 4 | 2 | 1 | 0 | 0 | 0 |
0 | 1 | X | 3 | 3 | 3 | 2 | 1 | 0 | 2 | X | 4 | X | 1 | 0 | 1 | 1 | 1 |
0 | 1 | 3 | X | 3 | X | 2 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 3 | X | 2 |
0 | 0 | 2 | X | 3 | 1 | 2 | X | 3 | 2 | 1 | 0 | 0 | 2 | X | X | 4 | 3 |
2 | 2 | 2 | 1 | 1 | 0 | 1 | 3 | X | X | 2 | 1 | 0 | 2 | X | X | 3 | X |
X | X | 3 | 1 | 1 | 0 | 1 | 3 | X | 4 | X | 1 | 0 | 1 | 3 | 3 | 3 | 1 |
3 | X | 3 | X | 1 | 0 | 2 | X | 4 | 3 | 1 | 1 | 0 | 1 | 2 | X | 1 | 0 |
1 | 2 | 4 | 3 | 2 | 0 | 3 | X | X | 1 | 0 | 0 | 1 | 2 | X | 2 | 1 | 1 |
0 | 1 | X | X | 2 | 1 | 3 | X | 4 | 2 | 1 | 0 | 1 | X | 2 | 1 | 0 | 1 |
0 | 1 | 3 | X | 2 | 1 | X | 3 | 3 | X | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 2 |
0 | 0 | 1 | 1 | 1 | 1 | 1 | 2 | X | 3 | 2 | 0 | 1 | 1 | 2 | 1 | 1 | 1 |
1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 2 | X | 2 | 2 | 2 | X | 3 | X | 1 | 2 |
X | 2 | 1 | 1 | X | 2 | 2 | 1 | 2 | 2 | X | 3 | X | 3 | X | 2 | 1 | 1 |
3 | X | 2 | 2 | 3 | X | 2 | X | 2 | 2 | 3 | X | 2 | 3 | 2 | 3 | 2 | 4 |
X | 2 | 2 | X | 2 | 1 | 2 | 1 | 2 | X | 2 | 1 | 1 | 1 | X | 2 | X | X |
```

Figure 60: Same board as 59, but solved with my algorithm

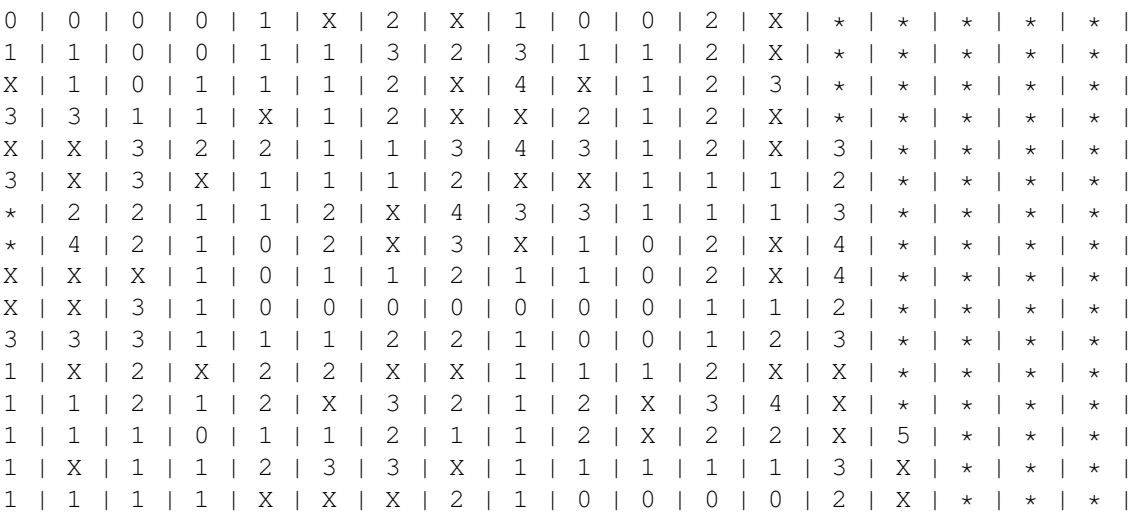

Figure 61: Results of running CSPStrategy

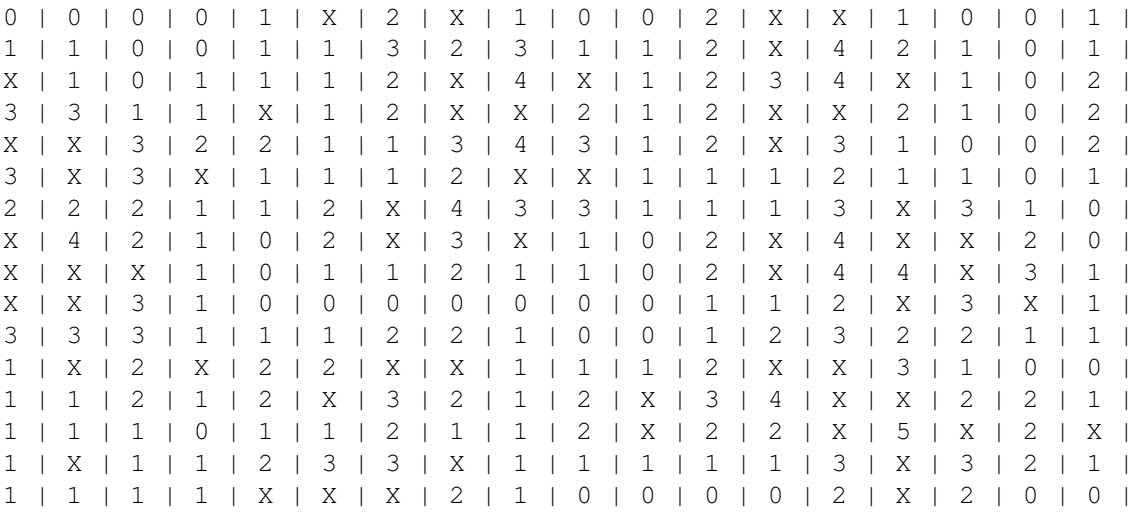

Figure 62: Same board as 61, but solved with my algorithm

The board on figure 59 is quite complex and I assume that even advanced player would have problems getting farther than Equation strategy without guessing, but it is possible and both CSPStrategy and my solver has managed it.
Board on figure 61 was solved by neither Equation strategy nor CSPStrategy, although

the latter was able to get farther. Again, I assume that solving this board without guessing is very difficult for human player, but it can be done as shown by my algorithm.
Each of the figures 59 through 62 shows only part of the board, where the algorithms got stuck. Complete boards, that are of standard Minesweeper size, can be found with other files distributed with this paper.

8.1.4 Computation time comparison

Since the main use of my algorithm is planned to be in real-time application, the computation time it requires should be as low as possible. In order to evaluate this, I've once again used the other two successful algorithms compatible with PGMS, the Equation strategy and CSPStrategy, as a comparison.

results for advanced game, 30x16 squares, 99 mines, 1000 runs

strategy	average runtime [milliseconds]	boards solved
Equation strategy	94.5	64
CSPStrategy	1.06	68
My CSP	206	70

results for intermediate game, 16x16 squares, 40 mines, 1000 runs

strategy	average runtime [milliseconds]	boards solved
Equation strategy	4.8	508
CSPStrategy	0.04	510
My CSP	9.3	518

results for beginner game, 9x9 squares, 10 mines, 1000 runs

strategy	average runtime [milliseconds]	boards solved
Equation strategy	0.16	790
CSPStrategy	0.025	790
My CSP	1.47	790

On a *beginner* board, my algorithm does terribly, taking 919% of the time Equation strategy needs on average and 5880% of CSPStrategy time. This hints at insufficient optimization of my algorithm that should be one of the main focuses of any consecutive work. For *intermediate* board, my solver takes 194% of time Equation strategy needs and 23250% of what CSPStrategy takes. On an *advanced* board, my strategy took 218% of the time Equation strategy needed, which is worse result than for *intermediate* board, and 19434% of the time CSPStrategy needed, which is smaller difference than for *intermediate* board, but still worse by a very large margin. The speed of CSPStrategy is even comparable to an SPS algorithm[6] a result that has surprised even Studholme, who is the author of CSPStrategy.
On a *beginner* board, all of the algorithms were able to solve the same amount of boards and the only advantage of using my algorithm with my generator is that it was built to work with it, but adapting CSPStrategy for this task should also be considered. On the *intermediate* and *advanced* boards, CSPStrategy was always able to solve more boards and my solver has solved more boards than CSPStrategy, which confirms the results from section (8.1.1)

8.2 Evaluation of the Minesweeper board generator

Unfortunately, there is no board generator compatible with PGMS, the only other board generator I know of (see section 3.4), which is implemented in C and completely standalone, uses only SPS and subset rule, which are some of the lower layers of Equation strategy (see section 3.3), so it should be able to consistently identify less solvable boards than Equation strategy. From experience of using it, it feels very responsive and the loading time for a new square was immediate even for *advanced* board.
To verify that a board is solvable, I've used all 3000 as an input for my solver and got a 100% success rate, meaning that all of the boards were evaluated as solvable.

results of my minefield generator, 1000 runs

square parameters	average runtime [milliseconds]	maximal runtime
Beginner 9x9 10mines	32.5	614
Intermediate 16x16 40mines	662	1589
Advanced 30x16 99mnines	7723	12990

The results for *beginner* and *intermediate* boards are satisfying. Being able to present the player with a board in 0.033, respectively 0.7 seconds, or alternatively, having maximal load times of 0.6 and 1.6 seconds is good, in my opinion.
The times for *advanced* board are much worse. 8 seconds on average and worst result of 13 seconds could lead into negative experience of the end user. It might be necessary to find a way to decrease the load times before using this algorithm to generate a board for human player.

9 Conclusion

I have implemented and evaluated an algorithm that is able to provide the player with a board that can be as difficult as rank 9, but still solvable. The time is, even for the most complex board, comparable to load times of small scale games. I have also presented an algorithm that can detect more solvable boards than other evaluated algorithms at the cost of higher computational complexity.
I have gathered and discussed many publications related to Minesweeper and provided a lot of context for understanding Minesweeper in general. I have focused heavily on examples with the use of handmade mock-ups of Minesweeper boards that should be easy to understand.

9.1 Future work

Since the board generator is fully operational and I have provided a working, albeit user unfriendly, GUI demonstration, the next step is designing an UI for the game engine. Since it takes several seconds to prepare the *advanced* board, the method used in GUI demo, where squares gradually change colour to signify that their position is now safe to start on, should be used. The *beginner* and *intermediate* boards seem to be generated quickly enough that the colour turning for safe squares seems unnecessary, but this might differ on different computers and the process should at least still run on

the background.
The algorithm itself is easily convertible into a multi threaded workload. A lot of computation is being done in millions of instances of PermWRepet and Combinations classes. The algorithm itself should be made faster for smaller boards, better support for methods that aren't CPU intensive, but less successful should be provided, for example the SPS is called after many of internal calculations are done, but those get thrown out in cases where SPS is successful. The minecount is only used when every other method fails, but it could help making *Analysing combinations of mines in a cluster* (6.1.9) more effective. Further study of CPSStrategy should be done, since its speed is very good, especially in comparison with other examined algorithms.
An algorithm focused on consistency problem should be implemented, possibly by focusing more on the blocks of squares my algorithm also uses, to further divide the size of the board. Minesweeper consistency is NP-complete and it would allow the algorithm to solve all of the other NP-complete problems. To enable this functionality an algorithm that reduces another NP-complete problem (probably SAT) to Minesweeper should be implemented, just as well as an algorithm that reduces Minesweeper consistency to another NP-complete problem, to aid with further study.

References

[1] Richard Kaye, *Infinite versions of Minesweeper are Turing complete*. School of Mathematics, The University of Birmingham, Birmingham, 2000

[2] Richard Kaye, *Minesweeper is NP-complete*. Mathematical Intelligencer, Birmingham, 2000

[3] Richard Kaye, *Some Minesweeper Configurations*. School of Mathematics, The University of Birmingham, Birmingham, 2007

[4] Minesweeper, *Authoritative Minesweeper*. URL `http://www.Minesweeper.info/index.html`, 2014

[5] Simon Tatham, *Simon Tatham's Portable Puzzle Collection*. URL `http://www.chiark.greenend.org.uk/~sgtatham/puzzles/doc/mines.html#mines`, 2014

[6] Chris Studholme, *Minesweeper as a constraint satisfaction problem*. 2000

[7] John D. Ramsdell. *Programmers Minesweeper* $PGMS$. URL `http://www.ccs.neu.edu/home/ramsdell/pgms/`

[8] Ken Bayer, Josh Snyder and Berthe Y. Choueiry, *An Interactive Constraint-Based Approach to Minesweeper*. Constraint Systems Laboratory, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 2006

[9] Stuart Russel, Peter Norvig, *Artificial Intelligence: A modern approach, 2nd edition*. 2003

[10] Stuart Russel, Peter Norvig, *Artificial Intelligence: A modern approach, 3nd edition*. 2010

[11] T Schiex, H Fargier, G Verfaillie, *Valued constraint satisfaction problems: Hard and easy problems*. IJCAI, Toulouse Cedex France, 1995

[12] D Frost, R Dechter, *In search of the best constraint satisfaction search*. Dept. of Information and Computer Science University of California, Irvine, AAAI, 1994

[13] Kasper Pedersen, *The complexity of Minesweeper and strategies for game playing*. Department of Computer Science, University of Warwick, 2004

[14] Marina Angelova Maznikova *Minesweeper Solver*. University of Mannheim

[15] Steven Givant, Paul Halmos *Introduction to Boolean Algebras*. Mills College, Department of Mathematics and Computer Science, 2009

[16] Stephen A. Cook *The Complexity of Theorem-Proving Procedures*. University of Toronto, 1971

[17] Allan Scott, Ulrike Stege, Iris van Rooij *Minesweeper May Not Be NP-Complete but Is Hard Nonetheless* The Mathematical Intelligencer, 33(4):5-17, 2011

[18] J. Kleinberg and E. Tardos *Algorithm Design* Addison Wesley, 2005

Appendices

Enclosed CD
There is an enclosed CD-ROM with every print of this thesis.
The contents of the CD-ROM are:
PDF version of this document
Netbeans project No.Guessing.Minesweeper, with the source files of the map generator ($MineMap.java$) and solver ($CSP\_FCH\_MRV.java$), also all the support classes and other code
Matlab files, $mainWindow.m$ will start the GUI demo