Approaches to creating solvable Minesweeper instances and providing assistance during game playing using constraint programming

Bachelor Thesis

Benedikt Simon Kunz
402097

15.07.2024

Supervisor: Prof. Dr. Benjamin Blankertz
Dr.- Ing. Stefan Fricke

Technische Universität Berlin
School of Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Neurotechnology

Abstract

Minesweeper is a well-known computer game in which all mines on a minefield must be found through logical reasoning. However, with a random arrangement of mines, as realized in the standard game, it may be necessary to guess. This can significantly reduce the fun of the game. This thesis presents approaches for creating solvable games, i.e., games that can be won through logical reasoning alone.
In a first step, a Minesweeper solver is developed. This is based on translating the game board into a Constraint Satisfaction Problem. The solution to this problem is then used to find mines. Using this solver, solvable Minesweeper games can be created. For the three standard sizes of Minesweeper (9 x 9, 10 mines; 16 x 16, 40 mines; 16 x 30, 99 mines) satisfying results are obtained. On average, it takes less than half a second to create solvable games.
In addition, it is also shown how a player can be given assistance during the game. The idea is as follows: In a first step, it is checked whether the player has already made any mistakes on the game board and is informed about them where necessary. In a second step, the player is shown where progress can be made. To do this, the easiest place to make progress on the game board is determined.

Zusammenfassung

Minesweeper ist ein bekanntes Computerspiel, bei welchem alle Minen auf einem Minenfeld durch logisches Denken gefunden werden müssen. Jedoch ist es bei einer zufälligen Anordnung der Minen, wie im Standardspiel umgesetzt, manchmal notwendig zu raten. Das kann den Spielspaß deutlich verringern. In dieser Arbeit werden Ansätze vorgestellt, um lösbare Spiele zu erstellen, also Spiele, die rein durch logisches Denken gewonnen werden können.
In einem ersten Schritt wird dafür ein Löser für Minesweeper entwickelt. Dieser basiert darauf, das Spielfeld in ein „Constraint Satisfaction Problem“ zu übersetzen. Die Lösung des Problems wird dann verwendet, um Minen zu finden.
Mithilfe dieses Lösers können schließlich lösbare Minesweeper-Spiele erstellt werden. Für die drei Standardgrößen von Minesweeper (9 x 9, 10 Minen; 16 x 16, 40 Minen; 16 x 30, 99 Minen) werden dabei zufriedenstellende Ergebnisse erzielt. Im Durchschnitt dauert es weniger als eine halbe Sekunde, um lösbare Spiele zu erstellen.
Darüber hinaus wird auch gezeigt, wie Spielern während des Spiels eine Hilfestellung gegeben werden kann. Die Idee ist wie folgt: In einem ersten Schritt wird geprüft, ob sich bereits Fehler auf dem Spielfeld befinden und der Spieler gegebenenfalls darauf hingewiesen. In einem zweiten Schritt wird dem Spieler gezeigt, wo weitere Fortschritte gemacht werden können. Dazu wird die Stelle auf dem Spielfeld ermittelt, an der dies am leichtesten möglich ist.

Contents

List of Algorithms

List of Figures

1 Introduction

Motivation

The computer game *Minesweeper* has gained significant popularity, being included in many versions of Windows since the 1990s. The simple rules make it easy for players to understand the game concept. Nevertheless, there are Minesweeper-specific problems that are computationally hard, such as the question whether a game is solvable [1]. It is not always possible to win a randomly generated game without guessing. Our work is intended to show approaches for preventing such situations, i.e., guessing during the game.

Aim of the thesis

This thesis focuses on the player's perspective and addresses two questions. On the one hand, approaches are to be shown on how solvable Minesweeper instances can be created. In this context, a game is considered solvable if it can be won without guessing. On the other hand, it is shown how players can be given assistance during the game. For both cases, the first step is to develop and optimize a Minesweeper solver that works with the help of *constraint programming*.

Literature review

Minesweeper appears in the literature from time to time. On the one hand, it is about the complexity of the Minesweeper-specific problems. For example, Sady Kane showed that the consistency problem — given a partially completed Minesweeper board, is there a possible arrangement of mines — is NP-complete [2]. In [3] and [1] it is shown that the inference problem — given a partially completed Minesweeper board, is there a cell that is provably safe — is coNP-complete. The solvability problem — given a Minesweeper board and the secret arrangement of mines, can the game be won without guessing — is also coNP-complete [1]. This means whether a random Minesweeper game is solvable is computationally hard to answer.
On the other hand, the literature is also about algorithms for solving Minesweeper [4, 5, 6, 7]. The majority of existing literature focuses on the question of how to increase the probability of winning in randomly generated Minesweeper games. However, [8] also addresses solvable Minesweeper instances. We will compare our implementation with this in chapter 5.

Structure of the thesis

Chapter 2 introduces the two foundations of this thesis. On the one hand, this is constraint programming and the associated Constraint Satisfaction Problem. On the other hand, the game Minesweeper is presented. Subsequently, the two concepts are combined, and it is demonstrated how Minesweeper can be solved with the help of constraint programming.

Chapter 3 first presents important definitions for the algorithms. Afterwards, a Minesweeper solver is developed and optimized. It is then demonstrated how solvable Minesweeper instances can be created. Finally, ideas are presented on how a player can be given assistance during the game.
Chapter 4 evaluates and compares different versions of the Minesweeper solver and the creation of solvable Minesweeper instances using the standard sizes of Minesweeper boards.
Chapter 5 interprets the results and shows which versions are most suitable as well as their limitations.

Acknowledgments

The Python[1] programming language was used to implement the algorithms. In particular, the packages NumPy[2] and python-constraint[3] were helpful. Inkscape[4] was utilized to create the graphics for Minesweeper. The presentation of the algorithms was realized with the help of the LaTeX package algorithm2e[5]. DeepL[6] was used for some linguistic adjustments.

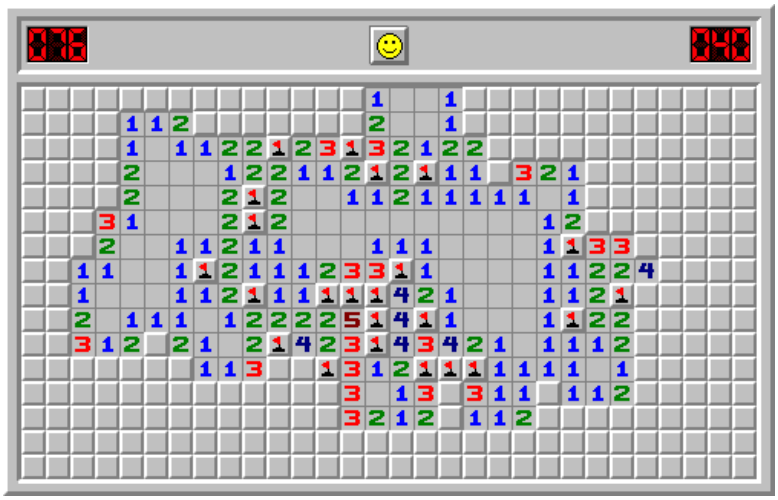


Figure 1.1: Minesweeper[7]

[1] https://www.python.org/
[2] https://numpy.org/
[3] https://pypi.org/project/python-constraint/
[4] https://inkscape.org/
[5] https://www.ctan.org/pkg/algorithm2e/
[6] https://www.deepl.com/
[7] Source: https://minesweeper.online/img/homepage/expert.png

2 Foundations

2.1 Constraint programming

2.1.1 Introduction

Constraint programming (CP) is a programming paradigm that is used to solve combinatorial search problems. Many techniques from artificial intelligence, computer science and operations research can be applied to this [9, p. 3]. CP has many facets and can be viewed from a variety of perspectives. For instance, one might consider the algorithms used to solve these search problems. In particular, backtracking and local search play a significant role. It is also possible to examine the computational complexity of these algorithms. In general, these problems are NP-hard. But if the constraints are restricted, for example, it may be possible to solve the problems efficiently [9, p. 4].
However, we restrict ourselves to formulating our problem as a Constraint Satisfaction Problem (CSP). This means, as a problem that can be given to a CSP solver so that we can then proceed with the result. This is the way in which CP is built into the algorithms. The actual CSP solver is considered as a black box. In other words, as a tool that can take an arbitrary CSP as input and returns a result.
In the context of this work, we look at how the game Minesweeper, or a state of Minesweeper, can be formulated as CSP and how to use this for algorithms. How CSP and Minesweeper can be brought together is explained in more detail in section 2.3. Constraint satisfaction basically describes finding matching values for the problem variables, where the constraints limit the possibilities. A simple example is the composition of a bicycle. It is necessary to select specific wheels, frames, pedals, and brakes. However, not every item is compatible with all others. Each bicycle component is represented by a variable, for each of which a specific model must be determined. The constraints describe which instances are compatible and which are not. So that all components of the resulting bike fit together. [9, p. 11].
The desired CSP solution can vary depending on the requirements. In the example of the bicycle, the question could be whether all parts are compatible with each other and the bicycle could therefore be composed. Another question is to find all the matching combinations. In our algorithms, the second question is particularly interesting, i.e., finding all solutions to a CSP.

2.1.2 Constraint Satisfaction Problem

The following definitions are based on [9, pp. 171, 246-247].

Definition 2.1 (Constraint Satisfaction Problem): A CSP $\mathcal{P}$ is a triple $\mathcal{P} = (X, D, C)$, where

- X is a finite set of n variables: $X = \{x_1, x_2, ..., x_n\}$
- D is a function that maps each variable x_i to the set D_i of possible values it can take: $D : X \to \{D_1, D_2, ..., D_n\};\ x_i \mapsto D_i$ for all $i \in \{1, ..., n\}$
- C is a finite set of t constraints (Definition 2.2), where the variables of each constraint c_j are in X: $C = \{c_1, c_2, ..., c_t\}$, $Var(c_j) \subseteq X$ for all $j \in \{1, ..., t\}$

D_i is called the corresponding *domain* of x_i.

Definition 2.2 (Constraint): A constraint c is a *relation* defined over the domains of a sequence $X(c)$ of k variables:

- $X(c)$ is the *sequence* of k variables of c: $X(c) = (x_{c_1}, x_{c_2}, ..., x_{c_k})$
- c is a relation over the domains: $c \subseteq D(x_{c_1}) \times D(x_{c_2}) \times ... \times D(x_{c_k})$
- $Var(c)$ is the *set* of variables of c: $\{x_{c_1}, x_{c_2}, ..., x_{c_k}\}$

This means, c is a subset of the Cartesian product of the domains of $X(c)$. $X(c)$ is also called the *constraint scope*. In contrast to the k-tuple $X(c)$, $Var(c)$ is the set of variables contained in it and thus $|Var(c)| \leq k$.

As described in [9, p. 84], a constraint c can be specified *intensionally* by a formula that the tuples in the relation must satisfy or *extensionally* by giving the set of tuples. In particular, if the number of tuples in the relation is infinite, the constraints must be specified as a formula. With the CSP solver we use for implementation, as with many others, the constraints are specified as formulas, which is also much more practical.

Definition 2.3 (Assignment): An assignment A of a CSP $\mathcal{P} = (X, D, C)$ is a mapping, that assigns to each variable $x \in X$ a value from its domain $D(x)$:

- $A : X \to \bigcup_{i=1}^{n} D_i; \quad x_i \mapsto v_i,\ v_i \in D_i$ for all $i \in \{1, ..., n\}$

Notation 2.1: For a sequence of variables $V = (x_{v_1}, x_{v_2}, ..., x_{v_m})$, we define:

- $A[V] \coloneqq (A(x_{v_1}), A(x_{v_2}), ..., A(x_{v_m}))$

Definition 2.4 (Constraint satisfaction): A constraint c is *satisfied* by an assignment A, if the values assigned to the variables fulfill the formula, respectively if the tuple obtained by assigning the values of A to the sequence $X(c)$, is contained in c:

- c is satisfied by A, if $A[X(c)] \in c$.

Definition 2.5 (Solution): A solution S to a CSP $\mathcal{P} = (X, D, C)$ is an assignment of $\mathcal{P}$, such that each $c \in C$ is satisfied.

In addition, we denote $sol(\mathcal{P})$ as the set of all possible solutions.

The definitions introduced are very general. In fact, we will only consider CSP where all variables in $\mathcal{P}$ have discrete and finite domains. $\mathcal{P}$ is then called a finite, discrete CSP. In this case, the set of possible assignments contains $|D_1| \cdot |D_2| \cdot ... \cdot |D_n|$ elements. If all domains have the same size, there are $|D_1|^n$ possible assignments.

The definitions will be illustrated with a short example.

Example 2.1:
We are looking for three natural numbers between 1 and 3, such that the sum of the numbers is less than or equal to 5 and the sum of the first two numbers is 3. As a CSP, this can be formulated as follows:

$\mathcal{P} = (X, D, C)$, where

- $X = \{x_1, x_2, x_3\}$
- $D : x_1 \mapsto \{1, 2, 3\},\ x_2 \mapsto \{1, 2, 3\},\ x_3 \mapsto \{1, 2, 3\}$
- $C = \{x_1 + x_2 + x_3 \leq 5,\ x_1 + x_2 = 3\}$

To illustrate, if we were to describe the constraint $c_2 = x_1 + x_2 = 3$ not as a formula, but as a set of tuples, it could be formulated as $c_2 = \{(1, 2), (2, 1)\}$, where $X(c_2) = (x_1, x_2)$.

The possible assignments are mappings, in which each variable x_1, x_2, x_3 is assigned to a value of $1, 2$ or 3. Thus, there are $3^3 = 27$ possible assignments.
A solution must also satisfy the constraints. The set of all solutions $sol(\mathcal{P})$ therefore consists of:

- $x_1 \mapsto 1,\ x_2 \mapsto 2,\ x_3 \mapsto 1$
- $x_1 \mapsto 1,\ x_2 \mapsto 2,\ x_3 \mapsto 2$
- $x_1 \mapsto 2,\ x_2 \mapsto 1,\ x_3 \mapsto 1$
- $x_1 \mapsto 2,\ x_2 \mapsto 1,\ x_3 \mapsto 2$

How exactly the set of solutions is computed is not considered here. We formulate our problem as a CSP $\mathcal{P}$ and a CSP solver returns $sol(\mathcal{P})$.

2.2 Minesweeper

2.2.1 Introduction

Minesweeper is a simple, single-player computer game that has been included on many versions of Windows. Mined-out, from 1983, is the earliest proven inspiration for Minesweeper. The initial release of Minesweeper was in 1990 in the first Windows Entertainment pack.[1] The goal of the game is to find the mines on a game board using logical reasoning, but also partly by luck. The following explanations are based on the Minesweeper help files.[2]

2.2.2 Playing Minesweeper

The game board of Minesweeper is a minefield, simulated by a grid of *cells* or *squares*. A cell can contain a mine, called a *mine cell* or simply a *mine*, or not contain a mine, called a *safe cell* or an *empty cell*. Initially, all cells are *covered*, i.e., a player does not know whether a cell is a mine or a safe cell.

The size of the game board and the mine density differs depending on the difficulty and can sometimes be customized. The three standard sizes are:

- Beginner: $9 \times 9 = 81$ cells, 10 mines
- Intermediate: $16 \times 16 = 256$ cells, 40 mines
- Expert: $16 \times 30 = 480$ cells, 99 mines

The most important property of a cell is its neighborhood. The *neighbors* of a cell are the maximum of eight surrounding cells. A cell at the border has five neighbors, while a cell in a corner has three.

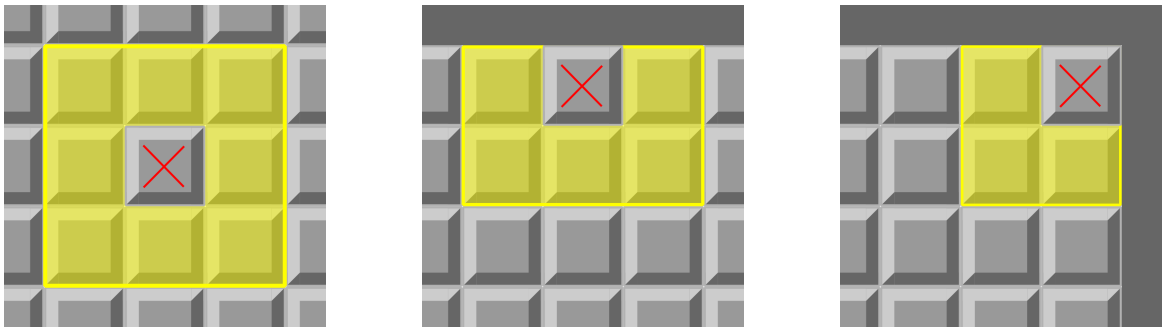


Figure 2.1: Neighbors of a cell

[1] https://www.minesweeper.info/wiki/Windows_Minesweeper (accessed May 28, 2024)
[2] https://www.minesweeper.info/downloads/WinmineHelpFiles.html

Moves

For each covered cell, the player has two options: *flagging* and *opening*.

- Flagging a cell means the cell will be marked by a flag. It is then called a *flagged cell*. A cell can also be unflagged and it becomes a covered cell again.
- Opening a cell means uncovering a cell to determine whether it is a mine or a safe cell. If it is a mine, the game is over; if it is a safe cell, the game continues and the number of mines in its neighborhood is displayed. The cell is then called an *open cell* or an *uncovered cell*. Opening a cell cannot be undone.

The displayed number of a cell, we also call the cell *type*. For example, if a cell has three mines in its neighbors, it has a cell type of 3.
There is a special feature for cells that have a cell type of 0, i.e. no mines in the neighborhood. If such a cell is opened, a blank cell is displayed instead of a 0. In addition, all its neighbors are automatically opened, since these are all safe cells.

(a) flagged cell (b) opened cell (c) blank cell

Figure 2.2: Cell examples

For the sake of completeness, another possible move is mentioned here. This is especially useful for players who play Minesweeper for time. It is possible to *clear* an open, safe cell. All covered cells in the cell's neighborhood, particularly those that are not flagged, are opened simultaneously. The prerequisite for clearing a cell, is that enough cells are flagged in its neighbors, i.e., at least the number of its cell type. However, if a mine is opened due to incorrect flagging, the game is over. Figure 2.3 shows how the cell outlined in red is cleared.

Goal

The goal of Minesweeper is to open all safe cells without hitting a single mine. The mines can be marked with a flag. This can be helpful during the game but is not necessary. The game ends and is won as soon as all safe cells have been opened or lost as soon as a mine is opened.

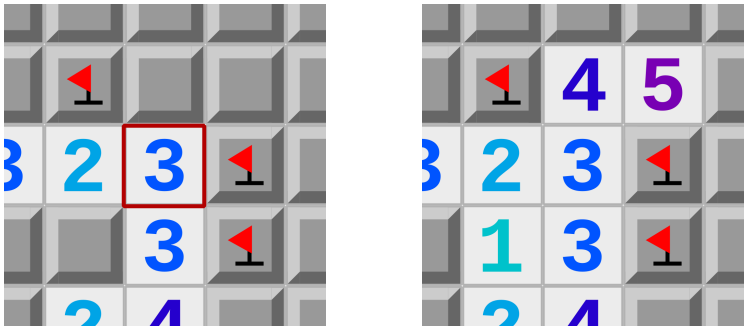


Figure 2.3: Clearing a cell

2.2.3 Winning Minesweeper

At the beginning of the game, all cells on the board are covered. The game begins as soon as the first cell is opened. To prevent a player from losing on the first move, the first cell is never a mine. The numbers displayed on the opened cells can be used to open further safe cells or to flag mines. New information can only be obtained by opening safe cells. Nevertheless, flagging mines can be helpful when playing. Sometimes, however, a player must guess and can only win the game by luck. This is illustrated by three examples. We always assume a perfect player who has not made any mistakes.

Easy Moves

Easy moves can be made in two situations. Firstly, if the number of flagged neighbors of a cell equals the cell type of this cell. Then all covered neighbors can be opened. Secondly, if the cell type minus the already flagged neighbors equals the number of covered neighbors. Then all covered neighbors can be flagged.
Consider the safe cell D in Figure 2.4, which has a cell type of 3. In its neighborhood, only the cells A, B and C are covered. Since there are three mines in the neighbors of D, the cells A, B and C must be mines and can be flagged. In this simple example, only one cell — cell D — needs to be considered to make progress.

Difficult Moves

It is slightly more difficult in Figure 2.5. Consider the cells G and J, both with a cell type of 1. It follows that only one of the three cells A, B and C can be a mine. Analogously, only one

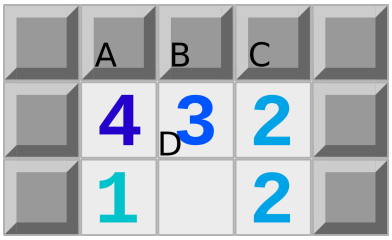


Figure 2.4: Easy moves

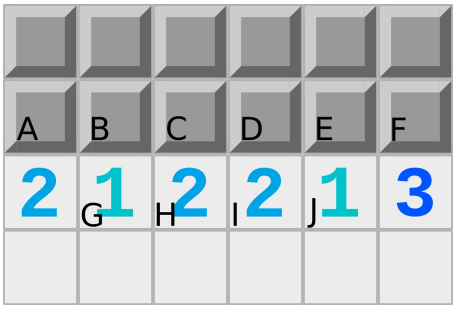


Figure 2.5: Difficult moves

of D, E and F is a mine. We will consider the cell H as an example. It has two mines in its neighbors. This implies two of the three cells B, C and D are mines. From cell G, we know that only one of the three cells A, B and C is mine and thus at most one of B and C is a mine. It follows directly, that D is a mine.
Using the same argumentation with the cells I and J, it follows that C is a mine. The remaining cells A, B, E and F are therefore safe cells. This means that the conditions for G, H, I and J are fulfilled and there is no other possible mine arrangement. In this example, the information from multiple cells had to be considered to make progress.

Unsolvable

Now consider the third example, Figure 2.6. One cell is already flagged correctly. Only one of the cells A and B, as well as one of the cells C and D is a mine. Due to the cell with a cell type of 3, the already flagged cell must be a mine. The question is, where are the other two mines? In fact, this cannot be determined clearly in this case. The two mines can be the cells A & C, A & D, B & C or B & D. To continue playing at this point, one must guess.

The third example also shows the meaning of solvable Minesweeper instances. A Minesweeper instance is solvable if a player can win the game without having to guess. A situation like that will not occur.

Figure 2.6: Unsolvable

2.3 Minesweeper and constraint programming

The two concepts presented, Minesweeper and constraint programming, will now be combined. This is quite easy to realize, as the opened cells with their cell types can be interpreted directly as constraints. However, a Minesweeper game is usually not directly translatable into a single CSP. This is because not all the information is available at the start of the game. New information is constantly being added if new cells are opened during playing the game. But every state of a game can be formulated as a CSP. If the state is solvable, new safe cells and mines can be identified by solving the CSP. The new state obtained in this way can again be formulated as a CSP and this process is repeated until the game is finished. This idea is not new and can be found in other works [4, 10, 6, 7].

Creation

The basic idea is straightforward. The covered cells that have opened cells in their neighbors form the variables. Since each cell is either a safe cell or a mine, the domain of these variables is set $\{0, 1\}$, where 0 stands for a safe cell and 1 for a mine.
The constraints can be formed from the opened cells that have covered cells in their neighbors. For an open cell c, the constraint can be formed as a formula as follows: The sum of the covered neighbors is equal to its cell type minus the number of flagged neighbors.

To illustrate this, consider Figure 2.7. The variables are the cells A to O. The constraints can be formed from the cells P to Z. We look at a few examples. The cell P describes the constraint: $A = 1$. Q describes the same constraint. $D + E + F + H + H = 5$ can be formed from the cell T. And cell Y forms the constraint $M + N + O = 3$. The complete CSP $\mathcal{P}$ of this state looks like that:

$\mathcal{P} = (X, D, C)$, where

- $X = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\}$
- $D : x \mapsto \{0, 1\}$
- $C = \{A = 1, A + B + C + D = 1, C + D + E = 1, D + E + F + G + H = 3, G + H + I = 3, H + I + J = 2, I + J + K + L + M = 4, L + M + N = 3, M + N + O = 3, N + O = 2\}$

Figure 2.7: Minesweeper and constraint programming

Solution

A CSP $\mathcal{P}$ formulated in this way is given to a CSP solver and the solution set $sol(\mathcal{P})$ is returned. Of the $2^{|X|}$ possible assignments, $sol(\mathcal{P})$ contains exactly those that satisfy all constraints. The solutions show how safe cells and mines can be distributed under the given constraints.
Receiving exactly one assignment, it is clear for every cell, whether it is a mine or safe. All variables that are assigned a 0 are safe cells and can be opened. All others that are assigned a 1 are mines and can be flagged.
If the solution set is empty, there is no assignment that satisfies all constraints. This can only happen if the constraints are incorrect and therefore the cell types on the board are incorrect. This does not happen here, as we always assume correct game boards.
In most cases, there will be multiple possible assignments. If there are variables that are always safe cells or are always mines in each assignment, these cells are clearly safe cells or mines and can be opened or flagged. However, if there is no variable that has the same value in each assignment, there is no unique solution, so the game cannot be solved without guessing. This is because it means that each variable can be either a mine or safe.

Fewer constraints

One important property should be mentioned. In a Minesweeper state, it is not always necessary to consider all constraints and variables to make progress. If one has a set of variables X and no constraints at all, then $sol(\mathcal{P})$ contains all possible $2^{|X|}$ assignments. However, adding just a few constraints can result in finding safe cells or mines. It should be emphasized that omitting constraints cannot lead to incorrect solutions. This is because omitting constraints can only increase the number of possible assignments, but never remove the actual assignment.

Consider again Figure 2.7 and the subset $X' = \{A, B, C, D\}$ of the variables. Without constraints, there are $2^4 = 16$ possible assignments. If the two constraints Q and R are included, i.e. $A = 1$ and $A+B+C+D = 1$, this results in a single solution: $\{A \mapsto 1, B \mapsto 0, C \mapsto 0, D \mapsto 0\}$. If only the constraint $A = 1$ is taken, the solution set contains the following assignments:

- $A \mapsto 1,\ B \mapsto 0,\ C \mapsto 0,\ D \mapsto 0$
- $A \mapsto 1,\ B \mapsto 0,\ C \mapsto 0,\ D \mapsto 1$
- $A \mapsto 1,\ B \mapsto 0,\ C \mapsto 1,\ D \mapsto 0$
- $A \mapsto 1,\ B \mapsto 0,\ C \mapsto 1,\ D \mapsto 1$
- $A \mapsto 1,\ B \mapsto 1,\ C \mapsto 0,\ D \mapsto 0$
- $A \mapsto 1,\ B \mapsto 1,\ C \mapsto 0,\ D \mapsto 1$
- $A \mapsto 1,\ B \mapsto 1,\ C \mapsto 1,\ D \mapsto 0$
- $A \mapsto 1,\ B \mapsto 1,\ C \mapsto 1,\ D \mapsto 1$

This already leads to the conclusion that $A = 1$, i.e. A is a mine. In this case, only a subset of the constraints was taken and progress could still be made. And it has not led to incorrect solutions.

If we only include the constraint $A + B + C + D = 1$, the possible assignments are:

- $A \mapsto 1,\ B \mapsto 0,\ C \mapsto 0,\ D \mapsto 0$
- $A \mapsto 0,\ B \mapsto 1,\ C \mapsto 0,\ D \mapsto 0$
- $A \mapsto 0,\ B \mapsto 0,\ C \mapsto 1,\ D \mapsto 0$
- $A \mapsto 0,\ B \mapsto 0,\ C \mapsto 0,\ D \mapsto 1$

In this case, no progress can be made. However, it does not lead to incorrect solutions. Including only a subset of the variables and constraints will become important later.

3 Methodology

3.1 Preliminary remarks

Programming

The Python[1] programming language is used to implement the algorithms. In addition to the standard libraries, the python-constraint[2] and NumPy[3] libraries are utilized. The CSP solver used in our implementation is based on a backtracking approach.

Programming language	`Python 3.10.12`
Additional libraries	`NumPy 1.26.4, python-constraint2 2.0.0b5`

Structure

The algorithm for creating solvable Minesweeper instances is based on a repeated call to a Minesweeper solver. For this reason, a Minesweeper solver is developed initially. To describe the algorithms, some Minesweeper-specific definitions are introduced first.

Pseudocode

The pseudocode is intended to explain the basic concept or idea behind the algorithms. It should be noted that the actual implementation may contain minor differences that are not shown in the pseudocode, to avoid unnecessary complexity. In most cases, the minor changes are also explained. Furthermore, for each pseudocode, the location of the corresponding code in the actual implementation is specified.
Some functions are available in multiple versions. The Minesweeper algorithm can then be given options to choose one of these versions. In such a case, it is specified how the options must be set to execute the corresponding function.

Other work

The algorithm was developed independently of [8], which also deals with solvable Minesweeper instances. The fundamental design differs, for example, in that an object-oriented approach is used in [8], where cells and constraints are implemented as objects. In contrast, our approach primarily operates with game boards. A brief comparison of speed is presented in chapter 5.

[1] https://www.python.org/
[2] https://pypi.org/project/python-constraint/
[3] https://numpy.org/

Nevertheless, many ideas of the algorithms are also found in other literature. This is noted in the appropriate places.

3.2 Definitions

The definitions 3.1–3.3, 3.9 are taken from [7], the definitions 3.5-3.8 are inspired by [7].

Unless otherwise specified, we always assume a fixed height h and a fixed width w, where $h, w \in \mathbb{N}^+$. h and w describe the height and width of the minefield or board, which consists of $h \cdot w$ cells.

Definition 3.1 (Cell): A *cell* c is a tuple, that represents its unique position on the board:

- $c = (x, y)$, where $x, y \in \mathbb{N}^+$, $1 \leq x \leq h$ and $1 \leq y \leq w$.

Two cells $c_1 = (x_1, y_2)$ and $c_2 = (x_2, y_2)$ are equal if and only if the x-value and the y-value are the same: $c_1 = c_2 \Leftrightarrow x_1 = x_2 \wedge y_1 = y_2$.

We start numbering from the top left. This means that the cell (1,1) is in the top left corner, the cell (1,w) is in the top right corner, (h,1) is located in the bottom left corner and the cell (h,w) is located in the bottom right corner.

Definition 3.2 (Cell Set): The cell set C is the set of all cells on the board:

- $C = \{(x, y) \mid x, y \in \mathbb{N}^+, 1 \leq x \leq h, 1 \leq y \leq w\}$.

As already mentioned, it applies: $|C| = h \cdot w$.

Definition 3.3 (Neighbors): The neighbors $N(c)$ of a cell $c = (x_0, y_0)$ is the set of surrounding cells:

- $N(c) = \{(x, y) \mid x_0 - 1 \leq x \leq x_0 + 1, y_0 - 1 \leq y \leq y_0 + 1, (x, y) \in C, (x, y) \neq (x_0, y_0)\}$.

This means $N|(c)| = 3$ for the four corner cells, $N|(c)| = 5$ for the other cells on the border and $|N(c)| = 8$ for all the remaining cells.
In addition to the neighbors, we also define the 2-neighbors.

Definition 3.4 (2-Neighbors): The 2-neighbors $N_2(c)$ of a cell $c = (x_0, y_0)$ is the set of cells defined by:

- $N_2(c) = \{(x, y) \mid x_0 - 2 \leq x \leq x_0 + 2, y_0 - 2 \leq y \leq y_0 + 2, (x, y) \in C, (x, y) \neq (x_0, y_0)\}$.

The 2-neighbors therefore also include the neighbors of the direct neighbors. It could also be defined as follows:

$$N_2(c) = \left(\bigcup_{n \in N(c)} N(n) \right) \setminus c$$

If we look at the board of a Minesweeper game, each cell is represented by exactly one symbol, such as a flag or a number. However, we first divide this representation into two properties, the cell *type* and the cell *status*.

- The cell type describes whether a cell is a mine or safe. If it is a safe cell, it also describes the number of mines in the neighbors.
- The cell status describes, whether a cell is *covered*, *flagged* or *opened*.

Definition 3.5 (Cell Type): The cell value $T(c)$ of each cell c is an integer between -1 and 8, i.e. $T(c) \in \{-1, 0, ..., 7, 8\}$ with the following meaning:

- $T(c) = -1$: c is a mine.
- $T(c) \geq 0$: c is a safe cell, where $T(c)$ is the number of mines in its neighbors, i.e. $T(c) = |\{n \in N(c) \mid T(n) = -1\}|$.

As indicated by the second bullet point in the definition, the mines, i.e. the cells where $T(c) = -1$ must first be determined before the cell type of the other cells can be calculated.

So mines are all cells for which $T(c) = -1$. The set of all mines M is defined by $M = \{c \in C \mid T(c) = -1\}$. All other cells are safe cells. The set of all safe cells S is defined by $S = \{c \in C \mid T(c) \geq 0\}$.

Definition 3.6 (Cell Status): The cell status $S(c)$ for each cell c is a number $S(c) \in \{-3, -2, 0\}$, with the following meanings:

- $S(c) = -3$: c is a flagged cell.
- $S(c) = -2$: c is a covered cell.
- $S(c) = 0$: c is an opened cell.

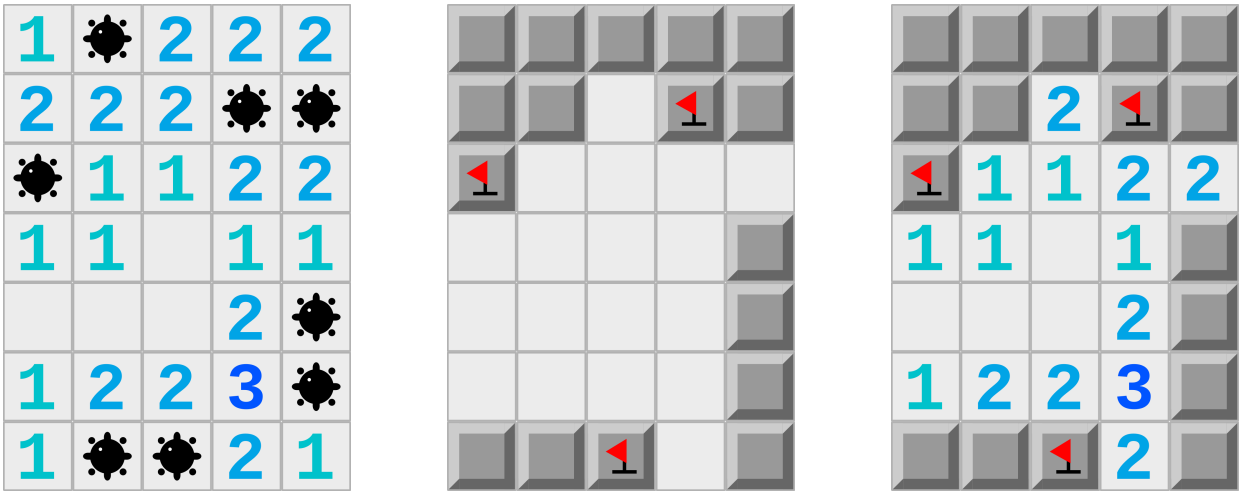
The values seem arbitrary at first, but we will come back to that in a moment.

Of course, a player does not know the cell type of each cell. This is because finding out which cells are safe is the goal of Minesweeper. However, a player knows the cell type of each opened cell. The two properties are used to define another property, the cell value, and this is exactly the information a player has.

Definition 3.7 (Cell Value): The cell value $V(c)$ for each cell c is an integer between -3 and 8, i.e. $V(c) \in \{-3, -2, ..., 7, 8\}$ and a combination of the cell type $T(c)$ and the cell status $S(c)$:

- $V(c) = \begin{cases} T(c) & \text{if } S(c) = 0 \\ S(c) & \text{otherwise} \end{cases}$

The three definitions presented, cell type $T(\cdot)$, cell status $S(\cdot)$ and cell value $V(\cdot)$, can be considered as a mapping for each cell, but also as a game board or a two-dimensional array T,



(a) cell type board (b) cell status board (c) cell value board

Figure 3.1: Cell properties

S and V, where $T(\cdot)$, $S(\cdot)$ and $V(\cdot)$ returns the value of the corresponding cell. This is how we consider these definitions.

In Figure 3.1, we see from left to right the boards T, S and V. The cell type board T can be seen as the underlying board. S as the board above. The cell value board V is the only board a player knows and therefore the only information a player has. It is a combination of T and S in such a way, that $T(c)$ is only visible for a cell c in V, if $S(c)$ is transparent, i.e., c is opened. A player therefore knows $T(c)$, if $S(c) = 0$. Initially, one can only see whether a cell is covered or flagged. Once a cell is opened, its cell type is visible.

The numbers for the cell status were therefore chosen to be unique in V. $V(c) = -3$ and $V(c) = -2$ describe a flagged or covered cell. For $V(c) \geq -1$, it is clear that the cell is open, where $V(c) = -1$ indicates a mine and $V(c) \geq 0$ a safe cell.

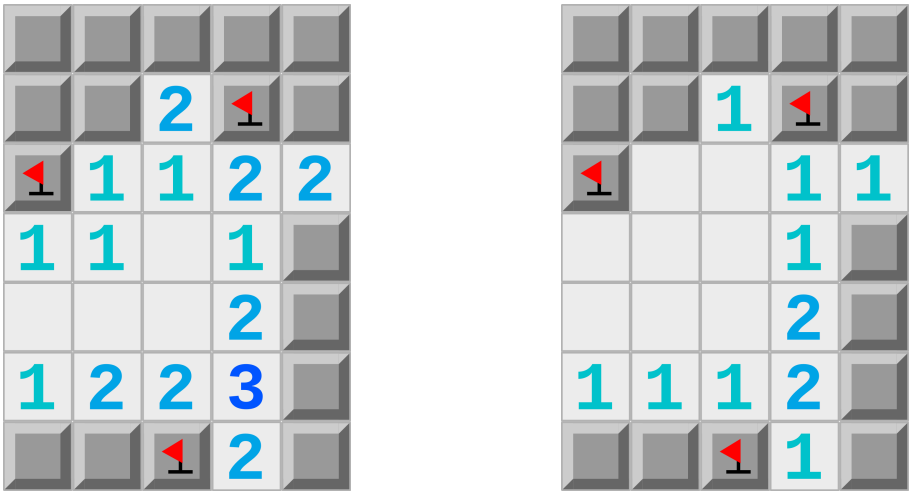
From the cell value, we define another parameter that is very important in the algorithms:

Definition 3.8 (Cell Sum): The cell sum $B(c)$ is a modification of the cell value $V(c)$. For each safe, open cell, the cell sum is the cell value minus the number of flagged cells in its neighbors. For all other cells, it is equal to the cell value:

- $B(c) = \begin{cases} V(c) - |\{n \in N(c) \mid V(n) = -3\}| & \text{if } V(c) \geq 0 \\ V(c) & \text{otherwise} \end{cases}$

Thus, for all open, safe cells, the cell sum describes the number of flags that are missing in its neighborhood if all mines are flagged. In this case, each safe cell would have a cell sum of zero at the end of the game.

Figure 3.2 shows the difference between cell value and cell sum.



(a) cell value board (b) cell sum board

Figure 3.2: Cell value and Cell sum

Particularly important for the algorithms is the set of unsolved cells.

Definition 3.9 (Unsolved Cell): An opened, safe cell c, i.e. $V(c) \geq 0$, is called an *unsolved* cell, if there are covered cells in its neighbors:

- c, where $V(c) \geq 0$, is an unsolved cell, if $|\{n \in N(c) \mid V(n) = -2\}| > 0$.

While we have focused mainly on the cells and the game boards so far, we will now define an actual Minesweeper instance.

Definition 3.10 (Minesweeper instance): A Minesweeper instance $\mathcal{M}$ is a 4-tuple $\mathcal{M} = (h, w, s, M)$, where

- h is the height: $h \in \mathbb{N}^+$
- w is the width: $w \in \mathbb{N}^+$
- s is the starting cell: $s = (x, y)$, where $x, y \in \mathbb{N}^+$, $1 \leq x \leq h$, $1 \leq y \leq w$
- M is the set of mines: $M = \{m_1, ..., m_k\}$, where $m = (x, y)$, $x, y \in \mathbb{N}^+$, $1 \leq x \leq h$, $1 \leq y \leq w$ for all $m \in M$

These are actually the only parameters needed to describe a Minesweeper instance $\mathcal{M}$. Everything else, such as the cell set or the cell type, can be determined from this. Therefore, the creation of solvable Minesweeper instances, means creating 4-tuples $\mathcal{M}$.

3.3 Solver

3.3.1 Basic concept

As mentioned above, we will develop the game creator in such a way that it calls a game solver. Therefore, we develop a game solver first. The basic idea is as this: The solver takes a Minesweeper instance $\mathcal{M}$ as input, i.e., a 4-tuple, computes the necessary game boards and tries to solve the instance. If this is possible, it returns a positive result, otherwise a negative one. The return value is therefore a Boolean value answering whether the Minesweeper instance $\mathcal{M}$ is solvable.

IsSolvable

Algorithm 1: *IsSolvable Version 1*

```
   Input  : M = (h, w, s, M): Minesweeper instance
   Output: solvable: boolean
 1 Function IsSolvable(M):
 2     U ← ∅                                          # set of unsolved cells
 3     T ← createCellTypeBoard(M)
 4     B ← createCellSumBoard(M)
 5     open s                                         # starting cell
 6     while progress is made do
 7         S_1, M_1 ← ConstraintSolver(U, B)          # S_1 = safe cells, M_1 = mines
 8         open the cells in S_1
 9         flag the cells in M_1
10     solvable ← CheckSolvability(B)                 # true or false
11     return solvable
```

Code	`GameSolver: is_solvable, solver_loop_v1`
Options	`EasyMoves: 0`

The *IsSolvable* function is the starting point of the solver. It first creates the set of unsolved cells and two game boards. The cell sum board B is the primary game board used by the algorithm. If a cell is opened, the cell type board is required, as the cell type of the opened cell is stored in it. The set of unsolved cells U could also be calculated from the cell sum board B, but we have decided to always calculate this set, because it is needed often. The set of unsolved cells can change when cells are opened or flagged. The changes in B and U are only shown implicitly in the pseudocode for the sake of simplicity and take place whenever cells are opened and flagged. We will come back to this in a moment.

If we look at the entire function again, the starting cell s is opened first. The cell sum board B, including the set of unsolved cells U, is then passed to the *ConstraintSolver*. The *ConstraintSolver* has the role of a perfect player. It takes the board as input and returns the sets of safe

cells and mines. These cells are then flagged or opened. This continues until the *ConstraintSolver* cannot find any new mines or safe cells. *IsSolvable* then checks whether the game board B is solved and returns the result.

Board creation: Back to the first lines of the algorithm. The two boards can be created very easily. For the cell type board, the mines are set to -1 as in the definition and for all other cells the number of mines in the neighborhood is calculated. The cell sum board B, i.e., the game board of the algorithm, initially only consists of covered cells. This means that the value of each cell on this board is set to -2.

Opening & Flagging: If a cell c is opened, the new cell sum $B(c)$ is calculated. The cell type is required for this, because the new cell sum is calculated as follows:
$B(c) = T(c) - |\{\, n \in N(c) \mid B(n) = -3\}|$. The cell sum is also adjusted if a cell is flagged: It applies $B(c) = -3$. Since this flagging has an effect on the cell sum of the neighbors, their cell sum is reduced by one. Furthermore, since the set of unsolved cells can change when opening and flagging, in both cases the cell and its neighbors are checked if they are newly added to the set of unsolved cells or if they are solved and thus removed from the set of unsolved cells.

Starting cell: Before *IsSolvable* reaches the while loop, the starting cell is opened first. To prevent losing the game immediately, it is often assumed, as in our case, that the starting cell is safe. In [7] this assumption is called "Safe first action rule". When the starting cell is opened, its cell sum is adjusted and the cell is added to the set of unsolved cells.

Check solvability: To determine whether the Minesweeper instance is solvable, the function *CheckSolvability* is called at the end of the algorithm. Here, two conditions are checked, whereby one of them must be fulfilled for the Minesweeper instance to be solvable. Either all mines are found and flagged, or all safe cells are found. Since each cell is either a safe cell or a mine, the other set can be determined by finding one of the two sets.

Figure 3.3 shows the cell sum board of a Minesweeper game. Five mines have been found and flagged. At this point, the *ConstraintSolver* cannot find any further safe cells or mines. Nevertheless, the instance may be solvable. If there are five mines in this game, all mines have been found. The covered cells are therefore safe and the game is solved. If the instance has

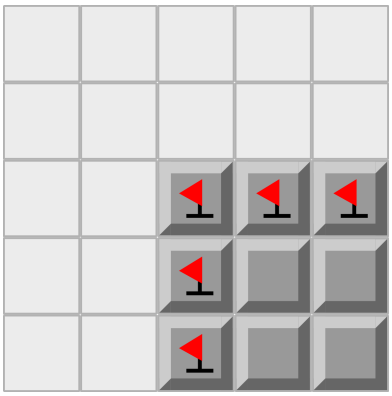


Figure 3.3: Check solvability

nine mines, then the covered cells are also mines. This means that all safe cells have been found and the instance is solvable. If there are 6–8 mines in the game, it is not solvable.

ConstraintSolver

Algorithm 2: *ConstraintSolver*

Input : U: set of unsolved cells, B: cell sum board
Output: S_1: set of safe cells, M_1: set of mines

1 **Function** *ConstraintSolver*(U, B)**:**
2 $V \leftarrow \emptyset$ `# set of variables`
3 $C \leftarrow \emptyset$ `# set of constraints`
4 **for** each $c \in U$ **do**
5 $var \leftarrow \{ n \mid n \in N(c) \land B(n) = -2 \}$ `# covered cells in neighbors`
6 V.add(var)
7 $con \leftarrow createConstraint(c, B, var)$
8 C.add(con)
9 $P \leftarrow createConstraintSatisfactionProblem(V, C)$
10 $Solutions \leftarrow getPossibleSolutions(P)$
11 $S_1 \leftarrow$ all cells that are always 0 in *Solutions*
12 $M_1 \leftarrow$ all cells that are always 1 in *Solutions*
13 **return** S_1, M_1

Code	`ConstraintSolver: cs_standard`
Options	`CSVersion: 0`

The *ConstraintSolver* solves a state of a Minesweeper game. It is given the cell sum board and a set of unsolved cells as input. The functionality is as described in section 2.3. The covered cells that have opened cells in their neighbors form the variables and are added to the set of variables V. The constraints are formed from the unsolved cells and added to the set of constraints C. For an unsolved cell c, a constraint is created as follows: $\sum N = B(c)$, where N is the set of covered neighbors, i.e., $N = \{ n \mid n \in N(c) \land B(n) = -2 \}$. Since $B(c)$ describes the number of missing flags, the constraint states the number of flags or mines that are missing within the set of covered neighbors.
Finally, a CSP P can be created from the set of variables and the set of constraints. Each variable is assigned the domain $\{0, 1\}$, where 0 represents a safe cell and 1 represents a mine. The solution set is then computed by a CSP solver. This set contains all possible solutions that satisfy all constraints. So, if a cell is a 1 in every solution, this cell must be a mine. If a cell is assigned to 0 in every solution, it must be a safe cell. The set of safe cells and mines is determined in this way and returned.
Neighbors: A side note should be given to the neighbors $N(c)$ such as in the fifth line. As soon as the size of the board is determined, the set of neighbors for each cell is also determined. It can therefore be calculated once at the beginning and can then be accessed again and again.

These two functions presented, provide a first version of the Minesweeper solver, which takes a Minesweeper instance as input and decides whether it is solvable. Next, we will look at optimizations that can be made to these functions.

3.3.2 Optimizations

IsSolvable

Algorithm 3: *IsSolvable Version 2*

Input : $\mathcal{M} = (h, w, s, M)$: Minesweeper instance
Output: *solvable*: boolean

```
1  Function IsSolvable(M):
2      U ← ∅                                          # set of unsolved cells
3      T ← createCellTypeBoard(M)
4      B ← createCellSumBoard(M)
5      open s                                          # starting cell
6      while progress is made do
7          EasyMoves(U, B, T)
8          if EasyMoves made no progress then
9              S_1, M_1 ← ConstraintSolver(U, B)       # S_1 = safe cells, M_1 = mines
10             open the cells in S_1
11             flag the cells in M_1
12     solvable ← CheckSolvability(B)                  # true or false
13     return solvable
```

Code	`GameSolver: is_solvable, solver_loop_v2`
Options	`EasyMoves: 1`

As mentioned in subsection 2.2.3, there are situations where easy moves can be made. We can take advantage of this. This idea can also be found in other works. For example, as "Primary Reasoning Strategy" in [7], as "Single Point Strategy" in [5] or as Step 1 of the CSPStrategy in [4]. We name the new function for these situations *EasyMoves*. It is integrated into *IsSolvable Version 2*. We always try to achieve progress with the *EasyMoves* function before the *ConstraintSolver* is called.
EasyMoves searches for unsolved cells c with one of two properties. Either $B(c) = 0$. Then all mines in the neighbors are already flagged. The remaining covered cells in the neighbors are therefore safe and can be opened. Or $B(c) = |C|$, where C is the set of covered cells in the neighbors of c, i.e., $C = \{\, n \mid n \in N(c) \wedge B(n) = -2\}$. c therefore has $|C|$ covered neighbors and there are still $|C|$ mines missing in the neighbors. This means that all cells in C are mines and can be flagged.

In these situations, only one constraint is required to obtain a unique solution. This means using *EasyMoves*, the *ConstraintSolver* is only called if at least two unsolved cells must be considered in a state to make progress.

An additional optimization can be found in the actual implementation of *EasyMoves*, which is not reflected in the pseudocode. To not determine unnecessarily often for each unsolved cell whether it is easily solvable, it is stored whether this has been checked. A new check only takes place if it could provide a different result. This can only happen for a cell if something has changed in its neighbors. Therefore, only if a cell is opened or flagged its neighbors are checked again.

Algorithm 4: *EasyMoves*

Input : U: set of unsolved cells, B: cell sum board, T: cell type board
Output: None

Function *EasyMoves*(U, B, T)**:**
  for each $c \in U$ **do**
    $C \leftarrow \{ n \mid n \in N(c) \land B(n) = -2 \}$ `# covered cells in neighbors`
    if $B(c) = 0$ **then**
      open each cell $c' \in C$
    else `#` $B(c) > 0$
      if $B(c) = |C|$ **then**
        flag each cell $c' \in C$

Code	`GameSolver: easy_moves, case_0_1, case_not_0`

ConstraintSolver

The next optimizations are related to the *ConstraintSolver*. Considering the unsolved cells in a state, it may be possible to divide the cells into different areas and compute them independently. This can provide a speed advantage by reducing the number of variables and the solution set. The unsolved cells, i.e. the constraints, can be divided into areas such that each variable only occurs in one area. This concept can also be found in other works, for example as "Frontier division" in [7].

Figure 3.4 shows how the unsolved cells can be divided into two areas, indicated by the yellow and red outlines. This is because the covered neighbors of all cells in an area, colored yellow and red, do not overlap. This means that the areas can be computed independently of each other and without affecting the others.

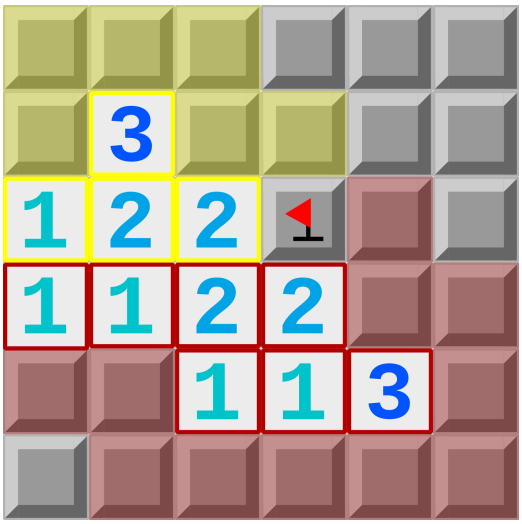


Figure 3.4: Division into areas

We call this method *DivideIntoAreas*. In a for loop, the variables of a cell are compared with the variables of each area. The variables of a cell are the covered cells in the neighbors, the variables of an area are all covered neighbors of all cells in the area. If the variables overlap, the cell is added to the corresponding area. Areas are merged if a cell can be assigned to more than one. If a cell is not assigned to an area, a new one is created. Each area is therefore a set of unsolved cells, whereby each cell only occurs in one area. *Areas* is the set of these sets and is returned. In the actual implementation, the variables of each area are also stored.

Algorithm 5: *DivideIntoAreas*

Input : U: set of unsolved cells, B: cell sum board
Output: *areas*: set of areas

```
1  Function DivideIntoAreas(U, B):
2      areas ← ∅
3      for each c ∈ U do
4          C ← { n | n ∈ N(c) ∧ B(n) = −2 }        # covered cells in neighbors
5          A ← ∅                                    # a set of areas
6          for each area ∈ areas do
7              if variables of area ∩ C ≠ ∅ then    # see explanation
8                  area.add(c)
9                  A.add(area)
10         if |A| > 1 then                          # c was added to more than 1 area
11             remove each area ∈ A from areas
12             newArea ← merge A
13             areas.add(newArea)
14         if |A| = 0 then                          # c was not added to any area
15             newArea ← ∅
16             newArea.add(c)
17             areas.add(newArea)
18     return areas
```

Code | `ConstraintSolver: divide_into_areas`

Two functions *ConstraintSolverExtension Version 1* and *ConstraintSolverExtension Version 2* have been created from this, which both include *DivideIntoAreas*.

In Version 1, the unsolved cells are divided by *DivideIntoAreas* as described and each area is computed with the original *ConstraintSolver*. Since each area is a subset of the unsolved cells, an area can be passed to the *ConstraintSolver* instead of the complete set of unsolved cells U.

In Version 2, the unsolved cells are also divided. In contrast to Version 1, however, the set of safe cells and mines is returned as soon as a safe cell or mine is found in any area and not all areas are computed. The idea behind this is to use *ConstraintSolver* less often and to first try *EasyMoves* again.

Algorithm 6: *ConstraintSolverExtension Version 1*

Input : U: set of unsolved cells, B: cell sum board
Output: S_1: set of safe cells, M_1: set of mines

Function *ConstraintSolverExtension*(U, B)**:**
    $S \leftarrow \emptyset$ `# S = set of safe cells`
    $M \leftarrow \emptyset$ `# M = set of mines`
    areas $\leftarrow$ *DivideIntoAreas*(U, B)
    for each *area* in *areas* **do**
        $S_1, M_1 \leftarrow$ *ConstraintSolver*$(area, B)$
        S.add(S_1)
        M.add(M_1)
    return S, M

Code	`ConstraintSolver: cs_extension_v1`
Options	`CSVersion: 1`

Algorithm 7: *ConstraintSolverExtension Version 2*

Input : U: set of unsolved cells, B: cell sum board
Output: S_1: set of safe cells, M_1: set of mines

Function *ConstraintSolverExtension*(U, B)**:**
    areas $\leftarrow$ *DivideIntoAreas*(U, B)
    shuffle *areas*
    for each *area* in *areas* **do**
        $S_1, M_1 \leftarrow$ *ConstraintSolver*$(area, B)$
        if $|S_1| + |M_1| > 0$ **then**
            return S_1, M_1
    return $\emptyset, \emptyset$

Code	`ConstraintSolver: cs_extension_v2`
Options	`CSVersion: 2`

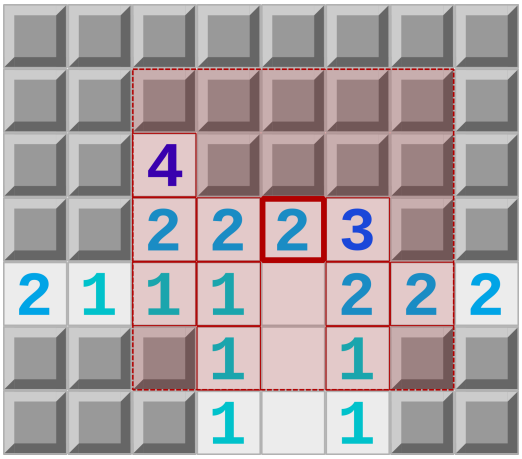


Figure 3.5: Random area

In chapter 4 we will see that often only 1-2 constraints are needed in a state to make progress. For this reason, the function *RandomTries* was created. The idea is to determine a random unsolved cell and to define all unsolved cells in its 2-neighbors as an area, which is passed to the *ConstraintSolver*. This often achieves connected constraints without dividing the entire game board into areas.
Figure 3.5 shows the 2-neighbors of a random cell. All unsolved cells in it are combined as an area and passed to the *ConstraintSolver*. This function is implemented as *RandomTries*. The random try can also be repeated n times if no safe cells or mines were found. If nothing could be found after n tries, *ConstraintSolver*, *ConstraintSolverExtension Version 1* or *ConstraintSolverExtension Version 2* is called, as previously described, with the complete set of unsolved cells.

Algorithm 8: *RandomTries*

Input : U: set of unsolved cells, B: cell sum board, n: number of tries
Output: S_1: set of safe cells, M_1: set of mines

Function *RandomTries*(U, B, n)**:**
  for $i \in \{1, ..., n\}$ **do**
    $c \leftarrow$ random unsolved cell
    $T \leftarrow N_2(c)$
    $S \leftarrow (U \cap T) \cup c$
    $S_1, M_1 \leftarrow$ *ConstraintSolver*(S, B)
    if $|S_1| + |M_1| > 0$ **then**
      return S_1, M_1
  return *ConstraintSolver*(U, B)

Code	`ConstraintSolver: cs_with_random_tries`
Options	`CSTries: 1, NTries: n (n` $\in \mathbb{N}^+$`)`

3.4 Creation

3.4.1 Basic concept

With the help of the presented Minesweeper solver, solvable Minesweeper instances can be created, whereby different variations are presented. The variants are implemented in different functions. Each of these functions takes as input h and w for the size of the board, n for the number of mines and s for the starting cell. The output of each function is a solvable Minesweeper instance.

At the beginning of each function, the set of possible mines P is calculated. It contains the cell set without the starting cell and its neighbors. If the starting cell or one of its neighbors were a mine, the board would not be solvable.
Minesweeper instances are then created in different ways and checked for solvability with the presented solver. If a solvable instance could be created according to the requirements of the input, it is returned.

3.4.2 Random

A first option, implemented as *GameCreationRandom*, is to randomly choose n mines from the set of possible mines and thus create a Minesweeper instance. If this instance can be solved, it is returned. Otherwise, the procedure is repeated until a solvable instance is found.
Each instance this function returns is therefore certainly solvable, as this only happens if the solver returns „solvable“.

Algorithm 9: *GameCreationRandom*

Input : h: height, w: width, n: number of mines, s: starting cell
Output: $\mathcal{M} = (h, w, M, s)$: Minesweeper instance

```
1 Function GameCreationRandom(h, w, n, s):
2     C ← CalculateCellSet
3     E ← N(s) ∪ {s}                          # excluded for mines
4     P ← C \ E                               # possible cells for mines
5     while no solvable instance created do
6         M ← choose n random cells from P
7         M ← (h, w, M, s)
8         if IsSolvable(M) then               # game solver call
9             return M
```

Code	`GameCreator: random_creation`
Options	`CreationMode: 0`

3.4.3 Iterative

Another option is to add the mines to the game iteratively, shown in *GameCreationIterative Version 1* and *GameCreationIterative Version 2*. For each new mine, the instance is checked whether it remains solvable. If the instance becomes unsolvable, the mine is rejected and another mine is tried. This process is repeated until the number n of required mines is reached. If there are no more cells in P, the set of possible mines, the process starts again.
It is important to note that a rejected mine is not permanently rejected. However, it is first added to the set of rejected mines R. If something changes on the game board, a rejected mine could be solved. This means that the rejected mines could be considered as new mines again, as soon as another mine has been added to the game.

Algorithm 10: *GameCreationIterative Version 1*

Input : h: height, w: width, n: number of mines, s: starting cell
Output: $\mathcal{M} = (h, w, M, s)$: Minesweeper instance

Function *GameCreationIterative*(h, w, n, s)**:**
- $C \leftarrow$ *CalculateCellSet*
- $E \leftarrow N(s) \cup \{s\}$ `# excluded for mines`
- $P \leftarrow C \setminus E$ `# possible cells for mines`
- $M \leftarrow \emptyset$ `# mines`
- $R \leftarrow \emptyset$ `# R: rejected mines`
- **while** $|M| < n$ **do**
  - **if** $P = \emptyset$ **then** `# restart game creation`
    - $M \leftarrow \emptyset$
    - $P \leftarrow C \setminus E$
    - $R \leftarrow \emptyset$
  - $m \leftarrow$ choose random cell from P
  - M.add(m)
  - P.remove(m)
  - $\mathcal{M} \leftarrow (h, w, M, s)$
  - **if** *IsSolvable*$(\mathcal{M})$ **then** `# game solver call`
    - $P \leftarrow P \cup R$
    - $R \leftarrow \emptyset$
  - **else**
    - M.remove(m)
    - R.add(m)
- **return** $\mathcal{M}$

Code	`GameCreator: iterative_creation_v1, try_create_game_v1`
Options	`CreationMode: 1`

GameCreationIterative Version 1 and *GameCreationIterative Version 2* differ when adding rejected mines to the set of possible mines. As soon as one new mine has been added to M in Version 1, all rejected mines are included to the set of possible mines again. In Version 2, all possible mines in P must been added to M at least once, before the rejected mines are included in P again.
These two functions also only return solvable instances. This is because the while loop ends if $|M| = n$. This means that the returned instance was solved in the last iteration.

Algorithm 11: *GameCreationIterative Version 2*

Input : h: height, w: width, n: number of mines, s: starting cell
Output: $\mathcal{M} = (h, w, M, s)$: Minesweeper instance

Function *GameCreationIterative*(h, w, n, s)**:**
    $C \leftarrow$ *CalculateCellSet*
    $E \leftarrow N(s) \cup \{s\}$ `# excluded for mines`
    $P \leftarrow C \setminus E$ `# possible cells for mines`
    $M \leftarrow \emptyset$ `# mines`
    $R \leftarrow \emptyset$ `# R: rejected mines`
    while $|M| < n$ **do**
        if $P = \emptyset$ **then**
            if tried all cells $r \in R$ after last mine was added **then**
                $M \leftarrow \emptyset$ `# restart game creation`
                $P \leftarrow C \setminus E$
                $R \leftarrow \emptyset$
            else
                $P \leftarrow R$
                $R \leftarrow \emptyset$
        $m \leftarrow$ choose random cell from P
        M.add(m)
        P.remove(m)
        $\mathcal{M} \leftarrow (h, w, M, s)$
        if not *IsSolvable*$(\mathcal{M})$ **then** `# game solver call`
            M.remove(m)
            R.add(m)
    return $\mathcal{M}$

Code	`GameCreator: iterative_creation_v2, try_create_game_v2`
Options	`CreationMode: 2`

3.5 Providing assistance

This section is intended to answer how a player can be given assistance during the game. The player's current game board, i.e., a cell value board, must be considered for that. In contrast to the creation of Minesweeper instances, there may be mistakes on the board.
For this reason, we present two principles on how assistance can be provided. A first option is to check the game board for mistakes that have been made. A second option is to point out where progress can be made.

3.5.1 Board check

As the game ends as soon as a mine is opened, mistakes can only be made due to incorrect flagging if the game is still running.

TooManyFlags

A first, simple mistake that can be made is flagging too many neighbors of a cell. This means that an open cell c exists for which the number of flagged neighbors is greater than its cell value, i.e., $V(c) < |\{\, n \mid n \in N(c) \land V(n) = -3\}|$. For this, *TooManyFlags* counts the number of flagged neighbors for each open cell and compares it with its cell value. If the number of flagged neighbors is too large, the cell is added to the set with incorrect cells F. In the implementation, one of these cells is then displayed to the player. This looks as follows:

```
The cell (1, 2) has too many flagged mines!
```

Algorithm 12: *TooManyFlags*

Input : V: cell value board
Output: F: set of cells with too many flags

Function *TooManyFlags*(V)**:**
  $F \leftarrow \emptyset$ `# set of cells with too many flags`
  for each cell c **do**
    if $V(c) \geq 0$ **then**
      $n_F \leftarrow \{\, n \mid n \in N(c) \land V(n) = -3\}$ `# flagged cells in neighbors`
      if $|n_F| > V(c)$ **then**
        F.add(c)
  return F

Code	`GameHelper: check_too_many_flags`

PotentiallyWrongFlags

Another mistake are potentially incorrectly flagged cells. This means flagged cells that cannot be safely determined as mines in the current state. These provide potential for further mistakes,

as incorrect conclusions can be drawn for safe cells. The reason these cells are only potentially flagged incorrectly is that further information in a later state could reveal the cells to be mines. The function *PotentiallyWrongFlags* implements this idea. First, a new board N is created from the existing board V, into which the flagged cells are not transferred and covered cells are added instead. All flagged cells in V are stored in F. The new board N is then solved, whereby all safe cells and mines are determined in this state. This works in the same way as the solver presented before. The only difference is that it does not solve an entire Minesweeper instance, but only the state of the board N. The set of all safe cells S in the state and the set of all mines M in the state are returned. Only the cells in M are safely mines in this state. The set of previously flagged cells F is compared with the mines M. If there is a cell in F that is not in M, it is potentially incorrectly flagged and is added to the set P of potentially incorrectly flagged cells.
In the implementation, the player is informed in a first step that there are potentially incorrectly flagged cells on the game board:

```
Potentially incorrect flags!
```

If this is not sufficient, a follow-up hint with a wrongly flagged cell is displayed:

```
The cell (7, 0) is potentially flagged incorrectly!
```

Algorithm 13: *PotentiallyWrongFlags*

Input : V: cell value board
Output: P: set of potentially wrong flags

```
1  Function PotentiallyWrongFlags(V):
2      N ← createEmptyBoard
3      F ← ∅                                          # set of flagged mines
4      for each cell c do
5          if V(c) = −3 then                          # c is a flagged cell
6              F.add(c)
7              N(c) ← −2                              # hidden cell
8          else
9              N(c) ← V(c)
10     M, S ← solve state N
11     P ← ∅                                          # potentially wrong flags
12     for each cell c ∈ F do
13         if c ∉ M then
14             P.add(c)
15     return P
```

Code	`GameHelper: check_potentially_wrong_flags; GameSolver: compute_mines`

3.5.2 Make progress

Help

The basic idea is to give the player the simplest possible assistance. *Help* therefore checks in a first step whether easy moves can be made. This is similar to *EasyMoves*. This means, it is checked whether there is a cell that can only be opened or flagged because of one other cell. If this is not possible, the next step is to check whether progress can be made at all. To do this, the current state is solved. This is similar to the function *ConstraintSolver*. If progress can be made, *EasiestSolvableCells* returns a cell that can be solved most easily, i.e., by the fewest other cells. Since one only ends up in *EasiestSolvableCells* if there are no easy moves, at least two cells are required to make progress in this case.

In the implementation, there are two different versions of *Help*. In the normal version, the player receives a cell in the first step that he can solve. This means a cell that is either a mine or safe:

```
Solvable cell:  (6, 0) [one cell required for solving]
```

If the player still cannot get any further, a message is displayed telling the player what kind of cell it is and why:

```
The cell (6, 0) is a mine because of (5, 1)!
```

However, Minesweeper can also be played without flagging. To win, all the safe cells must be opened. If no help should be given for mines, this can be set in the implementation. Assistance can still be given in this case. Since the player always receives help for a safe cell, the assistance is slightly adapted. In this case, the cells through which a new cell can be opened are named:

```
You can solve a cell with {(3, 1), (4, 1), (4, 0)}!
```

If the player does not get any further, the cell that can be opened is displayed in the second step:

```
The cell (5, 2) is safe because of {(3, 1), (4, 1), (4, 0)}!
```

EasiestSolvableCells

In a loop, all two-connected cells are passed to the *ConstraintSolver* and it is checked whether progress can be made. The cells are connected if at least one covered neighbor overlaps, i.e., at least one variable overlaps. If no progress can be made with all two-connected cells, n is increased to three and all three-connected cells are passed to the *ConstraintSolver*. This continues until a safe cell, or a mine is found. This will happen, as it was checked in *Help* that progress can be made in that state. What is not shown in the pseudocode, in the actual implementation the cells that are necessary to make progress are stored. These are used to display them to the player when required.

Algorithm 14: *Help*

Input : V: cell value board
Output: S: set of safe cells, M: set of mines

Function *Help*(V)**:**
    $S, M \leftarrow$ solve state V with easy moves
    if $|S| + |M| > 0$ **then**
        return S, M
    $S, M \leftarrow$ solve state V
    if $|S| + |M| = 0$ **then**
        return $\emptyset, \emptyset$
    $S, M \leftarrow$ *EasiestSolvableCells*(V)
    return S, M

Code	`GameSolver: help, compute_help`

Algorithm 15: *EasiestSolvableCells*

Input : V: cell value board
Output: S: set of safe cells, M: set of mines

Function *EasiestSolvableCells*(V)**:**
    $S \leftarrow \emptyset$
    $M \leftarrow \emptyset$
    $B \leftarrow$ *createCellSumBoard* from V
    $n \leftarrow 2$
    while $|S| + |M| = 0$ **do**
        for each set C of n connected cells **do**
            $S, M \leftarrow$ *ConstraintSolver*(C, B)
            if $|S| + |M| > 0$ **then**
                return S, M
        $n \leftarrow n + 1$

Code	`ConstraintSolver: compute_easiest_solvable_cells`

4 Results

4.1 Preliminary remarks

Programming

All tests were performed on a computer with the specifications listed below on the Ubuntu[1] operating system. For the graphics we used the library Matplotlib[2].

Specifications	`Processor: AMD Ryzen 7 4700U, Memory: 16.0 GiB`
OS	`Ubuntu 22.04`

Structure

In a first step, we use the Minesweeper solver to investigate the behavior of randomly generated games in terms of solvability. The solvability properties of randomly generated games also influence the creation of Minesweeper instances. In the next step, we compare different versions of the Minesweeper solver to determine the best parameters. In a last step, using this optimal solver, we compare the versions for creating solvable Minesweeper instances.

Providing assistance

There are no separate tests for providing assistance, since the same Minesweeper solver is used as for the creation of games.

4.2 Random games

In the following results, randomly generated Minesweeper instances were created and tested for solvability. For this purpose, a random starting cell and a random set of mines were determined for each individual game. The starting cell and its neighbors were excluded from the set of possible mines.

Mine density

Figure 4.1 shows the solvability ratio for the three standard sizes when the mine density changes. The tests were performed for each 0.01 change in mine density, with 5000 games generated for 9x9 and 16x16 and 1000 games generated for 16x30 for each density.

[1] https://ubuntu.com/
[2] https://matplotlib.org/

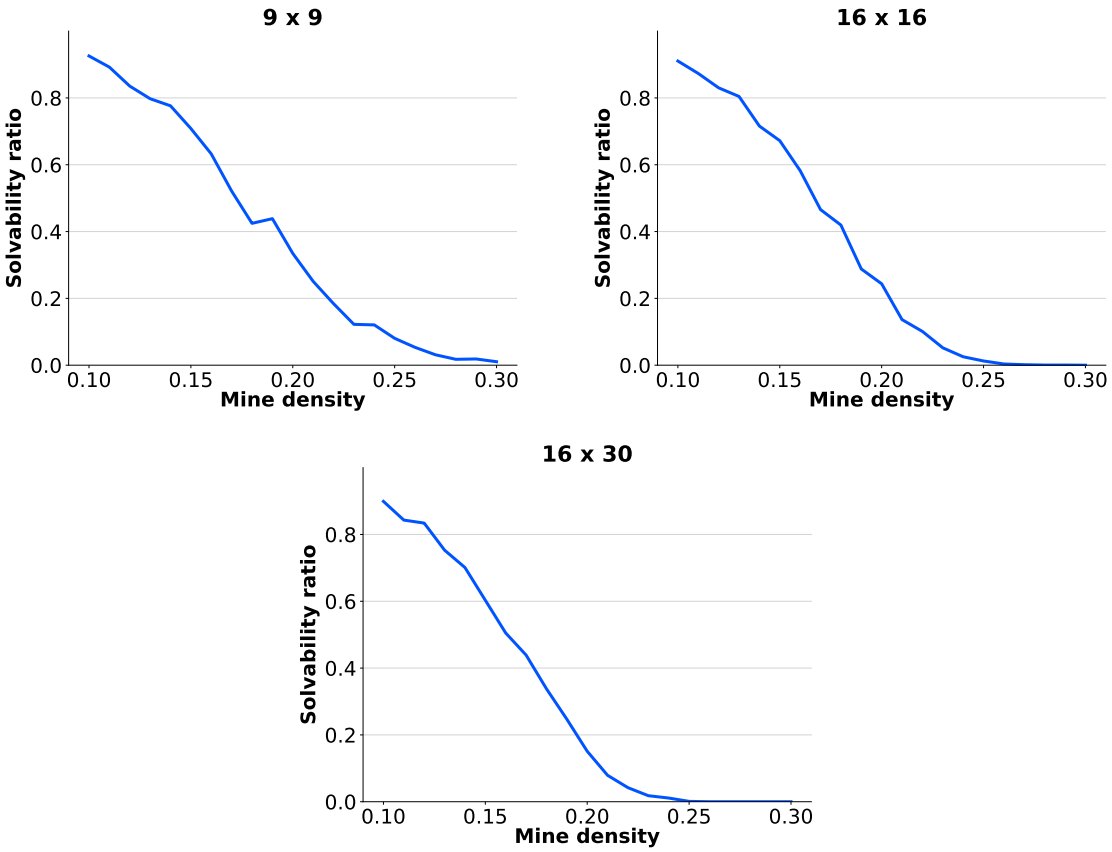


Figure 4.1: Solvability ratio and mine density

One can see that the ratio of solvable games decreases when the mine density increases. However, it can also be seen that the sizes behave differently. For example, for a mine density of 0.25, every 10th randomly created game is still solvable for the size 9x9, while for the size 16x30 almost no game is solvable.

Starting cell

Figure 4.2 shows the solvability ratio for different starting cells using the example of a 9x9 board. For each starting cell, 10000 games were created, with 14 random mines. This means a mine density of $14/81 \approx 0.17$ and according to Figure 4.1, every second game should be solvable on average.

The solvability ratio differs depending on the starting cell. A stating cell at the border and especially in the corners reduces the probability that a game with a random mine arrangement is solvable.

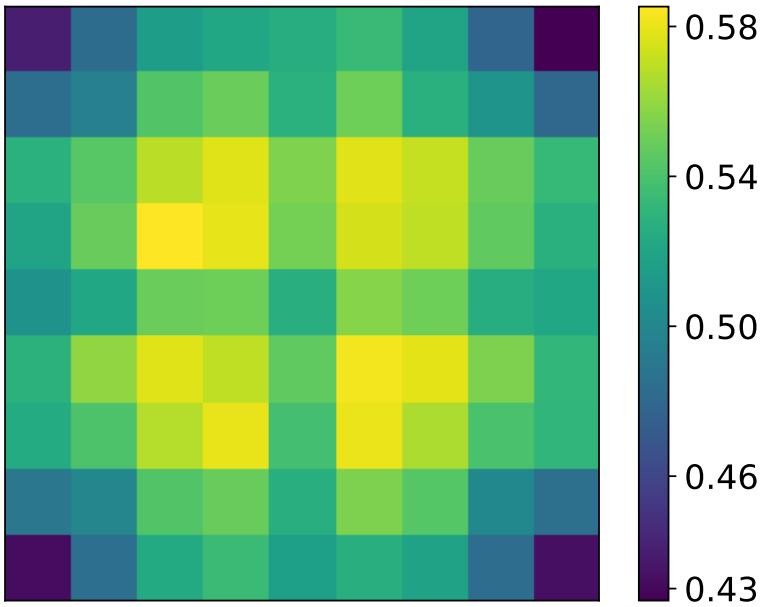


Figure 4.2: Solvability ratio and starting cell

Board size

In Figure 4.3, the solvability ratio is shown when the board size is changed while the mine density remains the same. This was evaluated for the quadratic board sizes 5x5 to 20x20. For the mine densities 0.1, 0.15 and 0.2, 10000 games were created for each point; for the mine density 0.25, 1000 games were created for each point. Trying to minimize the influence of the lower solvability ratio at the border, no starting cells were chosen at the border or one cell away from the border.

One can see that the ratio of solvable games decreases for larger boards, although the mine density remains the same. This is particularly noticeable at a higher mine density, here for example at 0.25.

Layout

The same test was performed for Figure 4.4, but here the layout was changed instead of the board size. Each board consists of 240 cells, which are arranged differently.

Although the number of cells and the mine density remains the same, the solvability ratio changes depending on the arrangement of the cells. Game boards are more likely to be solvable if the height and width of the game board do not differ greatly.

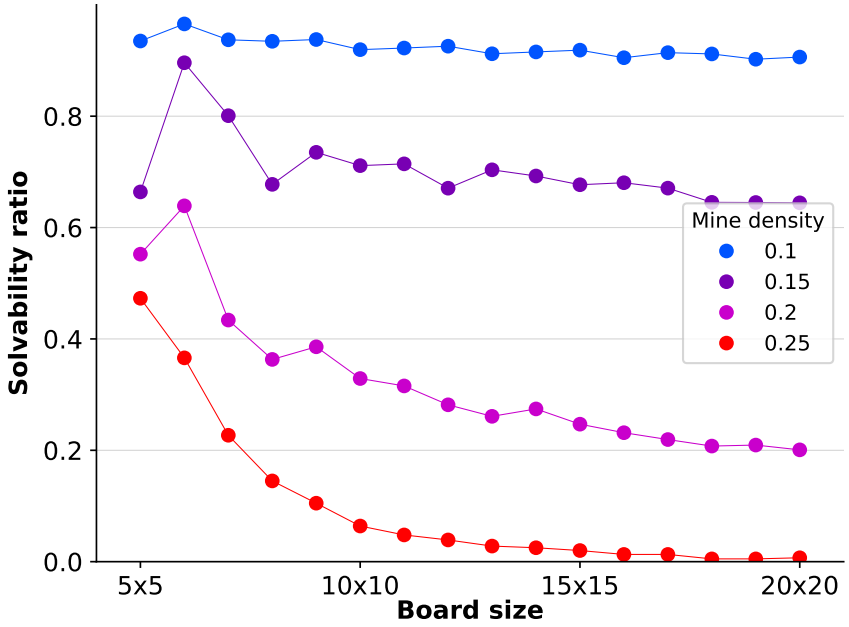


Figure 4.3: Solvability ratio and board size

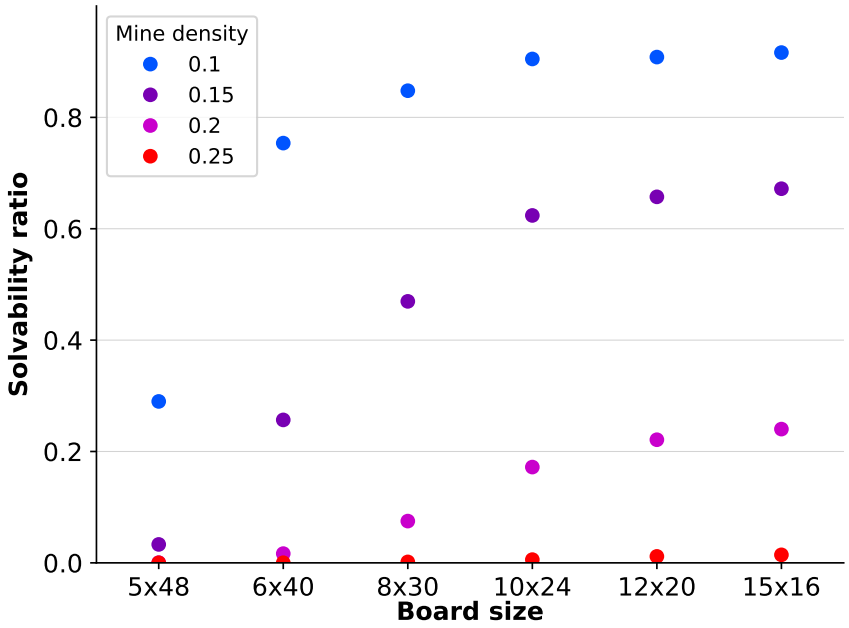


Figure 4.4: Solvability ratio and layout

4.3 Solver

The following results compare different versions of the Minesweeper solver, with the aim of determining the best parameters in terms of speed. For this purpose, Minesweeper instances are created with the function *GameCreationIterative Version 1*. To increase comparability, the seed for the selection of random mines was set to the same value in all versions and the same starting cells were used. This means that for each comparison, the same instances were created in the different versions.

(Beginner: 9x9 — 10 mines, Intermediate: 16x16 — 40 mines, Expert: 16x30 — 99 mines)

IsSolvable

Table 4.1 compares Version 1 of *IsSolvable* with Version 2, which included *EasyMoves*. 1000 instances were created for Beginner and Intermediate, 100 for Expert.

IsSolvable	Beginner		Intermediate		Expert	
	mean	std	mean	std	mean	std
Version 1	20.426	5.374	328.815	138.329	4065.760	5926.300
Version 2	4.304	1.306	61.513	29.949	933.905	1992.262

Table 4.1: Comparison of IsSolvable [in ms]

We see that Version 2 is significantly faster. In the following, we will therefore always use the Version 2 of *IsSolvable*, which includes *EasyMoves*.

Based on these results, we also took a look at how often *EasyMoves* can make progress and how often the *ConstraintSolver* is called in Version 2. These following percentages are the average of 1000 created games.
Creating Beginner boards, the *ConstraintSolver* was called in 1.70% of the while loop iterations. This means in the other cases *EasyMoves* could make progress. For Intermediate boards, the *ConstraintSolver* was called in 1.49% of the iterations, for the Expert boards in 2.86%.

ConstraintSolver

Table 4.2 compares the different versions of the *ConstraintSolver*:

- Version 0: *ConstraintSolver* (standard)
- Version 1: *ConstraintSolverExtension Version 1*
- Version 2: *ConstraintSolverExtension Version 2*

1000 games were created for each version.

Version 1 and 2 have no advantage over Version 0 when creating solvable Beginner and Intermediate boards, but there is no disadvantage either. Only when creating Expert Minesweeper instances, Versions 1 and 2 have a clear advantage. Versions 1 and 2 are almost equally fast, with version 1 having a greater standard deviation. Taking a closer look at the data, Version

Constraint-Solver	Beginner		Intermediate		Expert	
	mean	std	mean	std	mean	std
Version 0	4.188	1.506	55.774	19.136	2048.789	9074.232
Version 1	4.187	1.517	54.489	13.432	499.824	1009.466
Version 2	4.199	1.522	54.561	13.366	493.046	684.014

Table 4.2: Comparison of ConstraintSolver [in ms]

1 was faster than Version 2 in 651 of 1000 cases. We decided to continue with Version 1 in the following, which divides the game board into areas and computes all of them each time.

In addition, we took a look at how many cells need to be considered in the *ConstraintSolver* in order to make progress. Since we used Version 2 for this, it must be at least two cells, otherwise *EasyMoves* would have solved these cells.
With 100,000 calls to the *ConstraintSolver* for the creation of 9x9 boards of different mine densities, in over 96% of the cases existed at least one cell on the game board that was solvable with only two cells.
This is how the idea of *RandomTries* came about, to consider a random area of cells, which possibly results in progress.

RandomTries

Table 4.3 compares the function *RandomTries* with different numbers of tries. 0 means that *RandomTries* is not called and instead *ConstraintSolverExtension Version 1* is called directly. Again, 1000 games were created in each case.

number of tries	Beginner		Intermediate		Expert	
	mean	std	mean	std	mean	std
0	4.229	1.387	59.050	19.652	486.906	1164.28
1	4.267	1.466	57.822	15.448	439.027	840.991
2	4.239	1.387	57.854	16.456	447.085	960.898
3	4.251	1.432	57.849	16.560	449.323	994.698
5	4.246	1.400	57.878	16.113	441.858	819.018
10	4.261	1.464	57.918	16.776	446.180	891.690

Table 4.3: Comparison of RandomTries [in ms]

RandomTries does not bring any major improvements, but the creation of Expert Minesweeper instances is slightly faster. The version with one try performs best.

Summarized, to create solvable Minesweeper instances for the standard sizes, the following parameters of the Minesweeper solver performs best: *IsSolvable Version 2*, which uses *EasyMoves*, *ConstraintSolverExtension Version 1* and the use of the *RandomTries* with one try ($n = 1$).

4.4 Creation

The presented version of the Minesweeper solver is used to compare different versions for the creation of Minesweeper instances. For this purpose, 1000 games are created in each case. As the algorithms for creating the games differ, the games created in each version are different.

Table 4.4 compares the different versions of the game creation:

- Version 0: *GameCreationRandom*
- Version 1: *GameCreationIterative Version 1*
- Version 2: *GameCreationIterative Version 2*

Game creation	Beginner		Intermediate		Expert	
	mean	std	mean	std	mean	std
Version 0	0.865	0.732	4.119	5.493	257.296	1306.911
Version 1	4.354	1.228	57.105	24.737	464.756	606.081
Version 2	4.444	1.439	57.635	15.787	451.208	503.911

Table 4.4: Comparison of GameCreation [in ms]

The average creation time for a solvable instance is less than half a second for each version and configuration. To our surprise, *GameCreationRandom* always performs best. However, it has the largest standard deviation for Expert boards. The two versions of *GameCreationIterative* do not differ significantly.

In Figure 4.5 a comparison of the creation time for a changing mine density is shown for the three standard sizes. For each 0.01 change in density, 100 games were created.

One can note there is a point for each game size, a mine density, for which *GameCreationRandom* becomes significantly slower. However, the standard mine density is below this point for all sizes ($10/81 \approx 0.123$ for 9x9, $40/256 \approx 0.156$ for 16x16, $99/480 \approx 0.206$ for 16x30).

To better understand why *GameCreationRandom* is so fast, we took a look at the number of attempts *GameCreationRandom* needs on average to create solvable instances. The numbers again refer to 1000 created games.
For Beginner boards, *GameCreationRandom* needs an average of 1.21 attempts (std: 0.52) until a solvable board is found, for Intermediate instances 1.61 attempts (std: 1.04) are needed and for Expert boards 9.75 attempts (std: 9.53).
This makes *GameCreationRandom* the fastest version for all standard sizes.

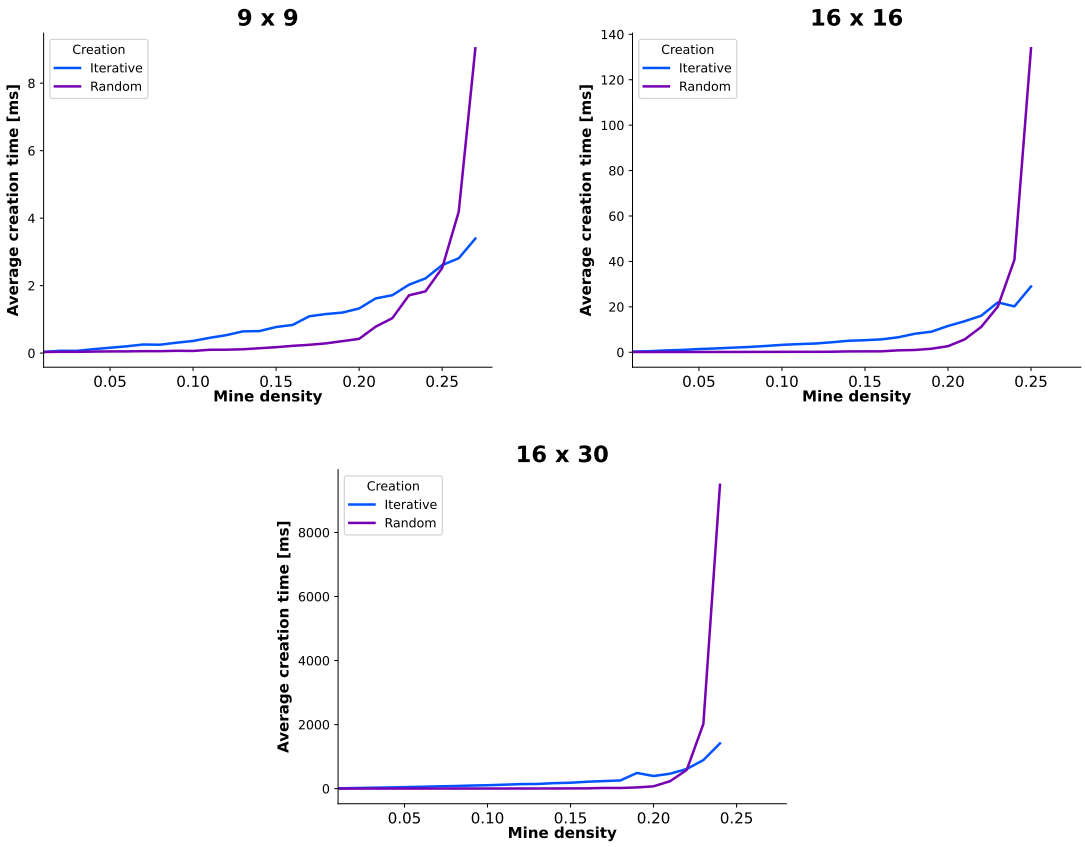


Figure 4.5: GameCreationRandom vs GameCreationIterative

While the standard sizes can all be created fast, Table 4.5 shows the average creation time with a higher mine density. For this, we chose the configuration 16x30, 170 mines, as this is available at Simon Tatham's Portable Puzzle Collection[3] for example. 100 games were created in each case.

Game creation	16x30, 170 mines mean	std
Version 1	7.822	13.493
Version 2	8.950	21.562

Table 4.5: Creation time with high mine density [in s]

The times required are significantly greater. According to these results, Version 1 is faster. *GameCreationRandom* cannot be evaluated for this mine density, as a random game occurs too rarely with this mine density and size. The creation time is too long.

[3] https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/mines.html

5 Discussion

5.1 Random games

The results show that all parameters of a Minesweeper instance (height and width, number of mines, starting cell) have an impact on its solvability. Larger game boards offer more possibilities for creating unsolvable mine arrangements. A higher mine density, or a starting cell in the corner or on the border, increases the probability of unsolvable mine arrangements. These parameters also have an influence on the creation of solvable Minesweeper instances. This is directly visible for *GameCreationRandom*. For parameters for which a solvable game is less likely, more tries are required to find a solvable instance. If more instances must be checked for solvability, more time is required on average.
The parameters also influence *GameCreationIterative*. Here, too, more iterations must be executed on average if there are more potential mines that make the game unsolvable. So, to compare different versions of Minesweeper solvers or instance creators, the parameters should be the same.

5.2 Solver

The results for the Minesweeper solver show that *EasyMoves* and *DivideIntoAreas* are the best optimizations. These are also the improvements that can be found in other works[4, 5, 7].
With *EasyMoves*, cells that are solvable with one constraint are solved directly by this function. On the one hand, this is very fast, on the other hand it also provides the advantage that the *ConstraintSolver* is called less frequently. And therefore, CSPs are created from the game board less frequently. It can also mean that the CSPs that are still being created are smaller and can be solved faster. How helpful this function is can also be seen from the result, that in only 1-3 % of the iterations of the while loop in *IsSolvable*, *EasyMoves* could not make any progress. For Minesweeper, this means that there are often easy-to-solve cells on the game board and that more difficult situations occur rarely.
DivideIntoAreas, which is built into *ConstraintSolverExtension Version 1* and *ConstraintSolverExtension Version 2*, can lead to a reduction in the size of the CSPs, as different areas can be considered independently of each other. As the number of assignments of a CSP can double with each variable, this can save a lot of time. For Beginner and Intermediate boards, there are no significant differences with or without *DivideIntoAreas*. It could be that smaller boards can be divided more rarely. It may also be, because with a lower mine density the *ConstraintSolver* is called less frequently and therefore the improvement is not as significant. Nevertheless, *DivideIntoAreas* does not bring any disadvantage for smaller boards.
RandomTries brings a minor advantage, at least for Expert instances. Although the improvement is small, the idea seems to work. If an area is randomly selected in which cells can be

solved, this saves time, because not the entire board is considered. Of course, *RandomTries* is also executed if a state is unsolvable. Then multiple tries are particularly unhelpful. This could be the reason, one random try provides the best results.
Overall, the fewer CSPs are created, the better for the time required.

5.3 Creation

We are satisfied with the average creation time for solvable Minesweeper instances. With milliseconds for Beginner and Intermediate instances and hundredths of a second for Expert instances, this can also be used in practice. Because this is the waiting time for a player, when he opens the first cell.
Table 5.1 shows the comparison with the implementation in [8]. Our implementations are faster for each size. However, it must also be said that we have tried to optimize our implementation for speed.

Game creation	Beginner		Intermediate		Expert	
	mean	std	mean	std	mean	std
Version 0	0.865	0.732	4.119	5.493	257.296	1306.911
Version 1	4.354	1.228	57.105	24.737	464.756	606.081
Version 2	4.444	1.439	57.635	15.787	451.208	503.911
[8]	32.5	-	662	-	7723	-

Table 5.1: Comparison of GameCreation [in ms]

Nevertheless, we were very surprised that *GameCreationRandom* is faster than *GameCreationIterative* for all standard sizes. This is, because the mine density is low enough for all standard sizes that only a few attempts are needed for a random mine arrangement to be solvable.

In practice, a Minesweeper instance creator should contain both, a function *GameCreationIterative* and the function *GameCreationRandom*. The idea is that the two functions can be combined. This means that a certain number of mines can always be added with *GameCreationRandom* first and further mines are then added with *GameCreationIterative*. The speed advantage of *GameCreationRandom* can be used at low mine densities, while *GameCreationIterative* is used at higher mine densities. However, the point at which *GameCreationRandom* becomes too slow differs depending on the board size and must be determined individually for each board size.

Whether *GameCreationIterative Version 1* or *GameCreationIterative Version 2* is better cannot be determined with our results. While we have assumed that Version 2 should be faster, as the rejected mines are excluded for a longer time, Version 1 was faster in the 16x30, 170 mines result. In order to make a reliable statement, further experiments would have to be performed. Eventually, the functions are too similar that no real difference can be detected. We would therefore decide for Version 1, because the mines are selected a bit more „randomly“.

The limits of the algorithm occur when the mine density is significantly increased. The results are good for the standard sizes, but if the mine density is increased, the required time increases

significantly. This makes sense, as the algorithm stays the same for each mine density. This means that more and more difficult game boards have to be solved and the probability that a new mine will make the game unsolvable increases.
To counteract this, further consideration would be needed of how solvable games can be created. For example, a different method could be used to add mines when the mine density is high, as implemented in [8], for example. This makes the games „less random“, but if it still results in interesting games, this is definitely a consideration of how the creation of Minesweeper instances can be improved.

5.4 Providing assistance

In contrast to creating a Minesweeper instance, the helper usually only has to solve one state. This means that the response time is also fast and it can be used with the functions described. We assume that new or casual players would definitely make use of this helper. It can also be used if a player has lost track of a large game board and needs a hint to cell solvable cell.

6 Conclusion

The thesis has shown on the one hand approaches how solvable Minesweeper instances can be created and on the other hand how players can be given assistance during the game. The solver is based on constraint programming, whereby a Constraint Satisfaction Problem (CSP) is created from the game board, which answers the question of safe cells and mines when it is solved. However, converting the complete game board into a CSP each time is very time intensive and so methods were presented to reduce the size of the CSPs or to consider only small parts of the game board. Different versions of this solver were compared and a best version was determined.

With this version, two methods for creating solvable Minesweeper instances were developed. On the one hand, an iterative approach was optimized, which adds mines to the game board one by one, and on the other hand, an approach that defines the entire set of mines at once and checks their solvability.
For the standard sizes, these methods were quite convincing. On average, only a few milliseconds to hundredths of a second are required to create solvable Minesweeper instances. This means that it can be used in practice. Surprisingly, the „non-iterative approach“ is the fastest for all standard sizes.
The algorithm reaches its limits when games with a high mine density are created. The time required then becomes significantly larger. This point can be addressed in future work. The question is therefore how the algorithm can be modified in case of high mine density, under the condition that still interesting games are created.

In order to provide a player with assistance, the following considerations were made. The input for a help function is the player's game board. This may already have incorrectly flagged cells. In a first step, the player can be informed about mistakes. If there are no mistakes on the board, the player can be shown a cell that can be solved, i.e., opened or flagged. To do this, the easiest cell to solve is chosen, i.e., a cell that requires the fewest other cells to be solved. If the player still cannot solve it, it is displayed whether it is a safe cell or a mine and also which other cells are used to derive this. As the help function is based on the Minesweeper solver presented above, this function also provides an answer in an acceptable amount of time and can be utilized.

Bibliography

[1] M. H. Group, D. Hendrickson, and A. Tockman, "Complexity of Planar Graph Orientation Consistency, Promise-Inference, and Uniqueness, with Applications to Minesweeper Variants," in *12th International Conference on Fun with Algorithms (FUN 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. Z. Broder and T. Tamir, Eds., vol. 291. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 25:1–25:20. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FUN.2024.25

[2] R. Kaye, "Minesweeper is np-complete," *Mathematical Intelligencer*, vol. 22, no. 2, pp. 9–15, 2000.

[3] A. Scott, U. Stege, and I. Rooij, "Minesweeper may not be np-complete but is hard nonetheless." *Mathematical Intelligencer*, vol. 33, no. 4, 2011.

[4] C. Studholme, "Minesweeper as a constraint satisfaction problem," *Unpublished project report*, 2000.

[5] K. Pedersen, "The complexity of minesweeper and strategies for game playing," *Project report, univ. Warwick*, 2004.

[6] Y. P. Sinha, P. Malviya, and R. K. Nayak, "Fast constraint satisfaction problem and learning-based algorithm for solving minesweeper," 2021. [Online]. Available: https://arxiv.org/abs/2105.04120

[7] C. Liu, S. Huang, G. Naying, M. N. A. Khalid, and H. Iida, "A solver of single-agent stochastic puzzle: A case study with minesweeper," *Knowledge-Based Systems*, vol. 246, p. 108630, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705122002842

[8] J. Cicvárek, "Algorithms for minesweeper game grid generation," Bachelor's Project, Czech Technical University, Prague, 2017.

[9] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[10] K. M. Bayer, J. Snyder, and B. Y. Choueiry, "An interactive constraint-based approach to minesweeper." in *AAAI*, 2006, pp. 1933–1934.