JOHANNES KEPLER
UNIVERSITY LINZ

Authors / Eingereicht von
Annika Schmidthaler
David Prinz
Reza Rajabi
Jessica Om
Aleksandar Stojanović
Matriculation number / Matrikelnummer
K12411307
K12412063
K12428264
K12411325

Institute / Department / Institut / Abteilung
Institute for Integrated Circuits and Quantum Computing

Lecture / LVA
Computational Complexity

December 22, 2025

Minesweeper

Group Project

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Contents

1 Introduction

1.1 Game Description & Rules

Minesweeper is a popular puzzle video game. The game features a grid of $x \times y$ tiles. Hidden throughout that field are multiple mines which the player needs to avoid. The game ends in a win if the player manages to clear all fields without detonating a single bomb.

Each tile within the grid can be either unopened, opened or flagged. Unopened tiles are blank and can be opened. Players are able to flag a cell to denote a possible mine location. Flagged cells are marked with a flag symbol on the grid and still considered as unopened.

Selecting a cell opens it. An open cell can either be a mine, which immediately ends the game and results in failure, a number that indicates the amount of mines that are horizontally, vertically or diagonally adjacent to it, or blank, in which case all non-mine cells neighbouring it will be revealed. A cell can have up to eight neighbours.

A game of Minesweeper begins by opening a cell. While playing, increasingly more information about the grid becomes known to the player which further aids in deducing the next safe cell to open. Furthermore, the remaining amount of mines is given to the

Figure 1: A game of Minesweeper

player. The mine count is calculated by subtracting the total number of mines by the number of flagged cells, thus also allowing the mine count to be negative.

To win at Minesweeper, the player has to clear all non-mine cells without opening a mine. There isn't a score count or time limit, however players' time to finish is being measured. Difficulty can be increased by adding more mines or starting with a larger game field.

After establishing the game rules and restrictions, we define our language **MSWP**: A field is in **MSWP** if there exists a layout of bombs such that the game field is valid, i.e. the numbers in the game field correctly correspond to the amount of neighbouring bombs.

1.2 Motivation

Choosing Minesweeper as this project's topic was appealing for a multitude of reasons. The primary ones of this paper are the wide-spread familiarity with the game and the topic being well-suited to the Computational Complexity lecture at JKU Linz.

Since Minesweeper has been bundled with numerous operating systems, millions of users have become deeply acquainted with its gameplay. Thus this paper should be more accessible to readers of a non-technical background.

To win at Minesweeper, players have to use logical reasoning to deduce which fields are safe and which are not. Such principals and strategies perfectly align with the subject matter covered in the lecture.

2 NP membership of Minesweeper

In this section of the paper we want to show that **MSWP** is in the problem class **NP**. So there exists a short certificate and a verifier that accepts that certificate in at most polynomial time. If that is the case, then our language is complete. If the verifier also rejects assignment which are not part of the language, then it is sound as well. Both of these requirements need to be fulfilled in order for our language to be part of **NP**.

For Minesweeper specifically we have to consider a placement of b on the game field G, such that every numbered cell correctly indicates the number of adjacent bombs (see section 2.4 for all rules that need to be fulfilled).

2.1 Certificate

The certificate in this case is a assignment of bombs to the game board grid, i.e. a configuration for each cell that tells us whether it's a bomb or not.

2.2 Accepting and Rejecting

To correctly verify the certificate, we need to (i) check the number of bombs of the certificate aligns with the number of bombs b of the input, (ii) for each numbered cell count the number of bombs in the neighbouring cells and verify that the count matches the value given in the cell.

MSWP fulfills the completeness criteria of **NP**, as if there is a Minesweeper game $MG \in$ **MSWP**, then a board configuration with b bombs exists, where all numbered cells are

consistent with the count of neighbouring bombs. Using this board placement as a certificate, the verifier will accept since all verification rules are fulfilled. As our input consists of the game field width n, game field height m and amount of bombs b (so $(n \cdot m) + 1$), the input size is at most polynomial ($O(n \cdot m)$). In the worst case (square grid, $n = m$) our runtime is $O(n^2)$.

The soundness of **MSWP** is present as well, because we reject every certificate that violates the game field rules set by us (see section 2.4). Thus it is not possible for a false certificate to be accepted.

As we have proven **MSWP** is complete and sound, thus we conclude that **MSWP** is a member of **NP**.

2.3 Encoding Minesweeper

Because of the relative simplicity and structured rule set of Minesweeper, we can construct logical formula based on those rules.

Input Because Minesweeper is played on a two-dimensional field, we have the game board width n and height m. Furthermore we also have the number of bombs b.

Output We want to know if the given configuration is a valid assignment of bombs.

Game Board Definition To effectively encode the game board, we assign a unique label F_iX to each cell on the game board. i is a natural number refers to a specific cell on the board. We start at the cell on the left-most upper corner of the game field and start counting up. $i \in \{1, \ldots, n \cdot m\}$.

X on the other hand specifies the amount of bombs that surround the current cell F_i. X ranges from 0 to 9, however we have chosen 9 to mean that a bomb is placed on that specific position. $X \in \{\underbrace{0,1,2,3,4,5,6,7,8}_{\text{neighbouring bombs}}, \underbrace{9}_{\text{bomb}}\}$.

Neighbours As previously established, a cell can have up to $k = 8$ neighbours. We encode each neighbour of the current cell F_i by n_k where $k \in \{1, \ldots, 8\}$. Each n_k has a truth value. If n_k is true, then that neighbour is a bomb. If it is false, then the neighbour isn't a bomb. We introduce this for easier notation when iterating over the neighbours.

2.4 Game Rules

Rule: Each field has at least one value $\bigwedge_{i=1}^{n \cdot m} (\bigvee_{x=0}^{9} F_i X)$

Rule: Each field has at most one value $\bigwedge_{i=1}^{n \cdot m} \bigwedge_{0 \leq x_1 < x_2 \leq 9} (\overline{F_{ij} X_1} \vee \overline{F_{ij} X_2})$

Rule: Across the entire game board there are at least b bombs $j = n \cdot m - b + 1$

$\binom{n \cdot m}{j}$ clauses - j literals each

$$\bigwedge_{1 \leq i_1 < i_2 < \ldots < ij \leq n \cdot m} (F_{i1}9 \vee F_{i2}9 \vee \ldots \vee F_{ij}9)$$

Rule: Across the entire game board there are at most b bombs $\binom{n \cdot m}{b+1}$ clauses - $b + 1$ literals each

$$\bigwedge_{1 \leq i_1 < i_2 < \ldots < ib < ib+1 \leq n \cdot m} (\overline{F_{i1}9} \vee \overline{F_{i2}9} \vee \ldots \vee \overline{F_{ib}9} \vee \overline{F_{ib+1}9})$$

Rule: if a field has value 0, then no neighbours are a bomb $n \cdot m \cdot 8$ clauses - 2 literals each

$$\bigwedge_{i=1}^{n \cdot m} \bigwedge_{k=1}^{8} (\overline{F_i 0} \vee \overline{n_k})$$

Rule: if a field has value z, then exactly z neighbours are bombs $1 \leq z \leq 7$

at least z bombs:

$j = 8 - z + 1$

$\bigwedge_{i=1}^{n \cdot m} \bigwedge_{1 \leq k_1 < k_2 < \ldots < kj \leq 8} (\overline{F_i X} \vee n_{k1} \vee \ldots \vee n_{kj})$

$n \cdot m \cdot \binom{8}{j}$ clauses

$j + 1$ literals each

at most z bombs:

$j = z + 1$

$\bigwedge_{i=1}^{n \cdot m} \bigwedge_{1 \leq k_1 < k_2 < \ldots < kj \leq 8} (\overline{F_i X} \vee \overline{n_{k1}} \vee \ldots \vee \overline{n_{kj}})$

$n \cdot m \cdot \binom{8}{j}$ clauses

$j + 1$ literals each

Rule: if a field has value 8, then all neighbours are a bombs $n \cdot m \cdot 8$ clauses 2 literals each

$\bigwedge_{i=1}^{n \cdot m} \bigwedge_{k=1}^{8} (\overline{F_i 8} \vee n_k)$

3 Karp Reduction from 3-SAT

3.1 SAT $\leq_p$ MSWP

To encode a **SAT** formula F with n variables and m clauses into an instance of **MSWP** M, we define following structures which function as building blocks. (see section 3.4).

3.2 Alphabet & Notation

First of all we introduce an alphabet that will help us later with the verification. The alphabet encodes all possible values a cell within the game field may have. A cell could be number n without zero (same behaviour as in Minesweeper), be blank (defined here as 0), have a predefined bomb, an input-dependent bomb, an input-dependent safe cell, a variable (unopened cell) or a cell without a value. Basically we define $\alpha = \{n, 0, \cdot, \cdot, \times, ?, \cdot\}$.

As the following chapters will frequently use the alphabet in multiple figures, we created a reference table for the readers' convenience.

Symbole	Definition
n	denotes how many bombs are in the neighbourhood of cell
0	blank cell
$\cdot$	predefined bombs
$\cdot$	input-dependent bomb
$\times$	input-dependent safe cell
?	value of variable (can be true: $\times$ or false: $\cdot$)
$\cdot$	input-dependent cell (can become $\times$ or $\cdot$ depending on variable value)

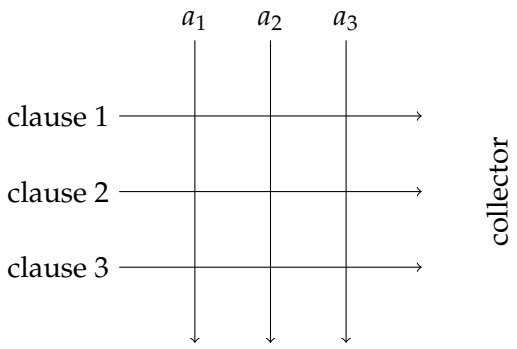
Table 1: Notation Table

3.3 Evaluation

The method to compute our Minesweeper field into a logical formula goes as such:

- We divide the game field into rows and columns
- We form clauses via the rows, by going from left to right
- We compute variables by going from top to bottom for each column
- To aid with our computation, we use the help of multiple components which alter the input stream of our rows and columns

3.4 Components

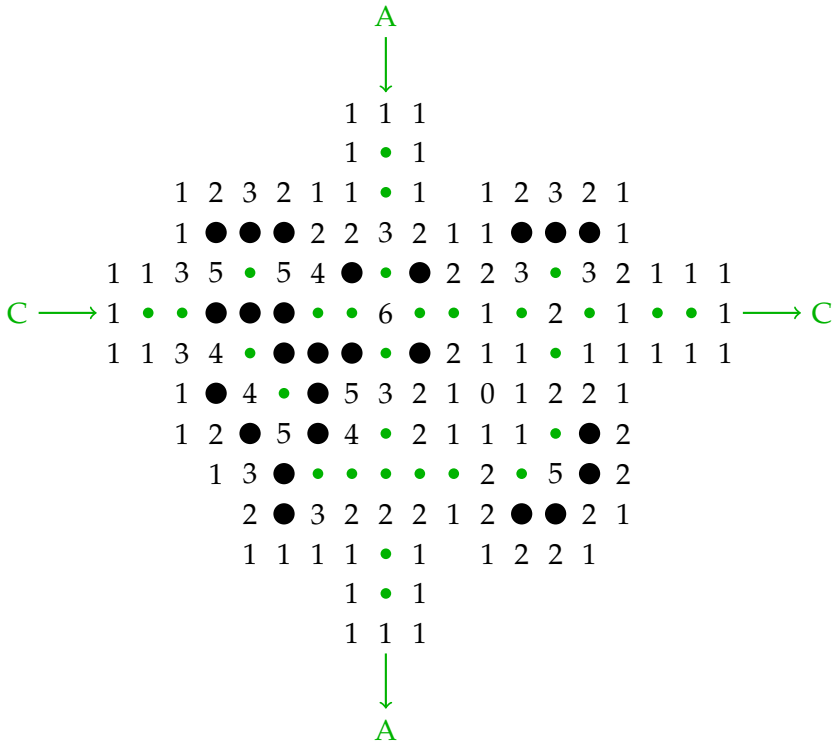

a_1
a_2
a_3
clause 1
clause 2
clause 3
collector

INPUT STREAM This input stream goes from top to bottom and evaluates each variable (? and $\cdot$) and sets its according value ($\times$ or $\cdot$). (See Table 1)

CLAUSE STREAM Represent a clause within our formula. It starts out as false. When we encounter a variable we check its truth value. If it evaluates to true, then our clause becomes true. Should the variable be false, but the clause's previous state was true, then the clause still stays true. So we use a logical or on the clause itself and the variable in the input stream $\underbrace{(x \vee y \vee \ldots)}_{\text{Clause}} \vee x$.

In simple terms: The Clause Stream "carries" the state of the clause from left to right, and as soon as an input satisfies the clause, the clause becomes true for the rest of the stream.

CROSSING This is used when the Clause Stream and input stream meet, but the variable of the Input Stream is not part of the clause. This component causes for the variable in the Input Stream to be ignored by the Clause Stream, i.e. the Clause Stream will not use the variable. The Clause Stream and Input Stream remain unchanged.



OR A simple disjunction of the input stream of the clause and the variable input stream. The clause becomes true if it either was already true or the current literal is true. Functionally identical to a logical OR-Gate.

SPLITTER This takes a variable input stream and splits it into two outputs with the same value. This is primarily used to aid us with the **OR** component, as otherwise we'd only have the disjunction of the clause and the literal, but this way we get to compute the variable input stream further and keep the calculation from the **OR** component. Thus we always use this before an **OR**.

TURN STREAM Turns a horizontal stream into a vertical one. Needed as some components stop computing the pure input stream, but as we often still need to further process the input stream, we use this component. This module, alongside **SPLITTER**, is necessary when using the **OR** component. **TURN STREAM** main functionality is that it brings both

the Clause Stream and Input Stream together. Also the Crossing and **OR** components expect that the Input Stream is vertical.

HORIZONTAL/VERTICAL OFFSET STREAM Offsets the variable input stream horizontally or vertically. We introduce this component as we often need to align the input stream after certain operations and as the input stream only repeats on every third cell.

COLLECTOR This component signifies the end of a clause. All collectors have to evaluate to true, as otherwise the formula will not be satisfiable and the game field will thus also be inconsistant.

END Signifies the end of the variable input stream.

3.5 Constructing SAT Formula

After explaining every component within our system, we want to use them to construct a SAT formula ϕ. To do that we first take a look at the Input Stream. The Input Stream receives an input-dependent cell and then, depending on the value of the variable, we can determine if the input x_i is a safe cell (true) or a bomb (false).

Next up we analyse the Clause Stream. As defined in section 3.4 the Clause Stream component determines if a clause evaluate to true or not, by analysing the next literal or by taking the clauses previous state. Clauses are intialised to *false* by default.

Finally each clause ends up in a Collector which enforces clauses to evaluate to true. (We achieve a similar feat to that of 8.7 of the lecture script, as we have broken a large formula up into the input, consistancy and termination parts).

As is immediately obvious, a constucted MSWP formula directly emulates a given SAT formula, from the literals to the clauses. Therefore any model of the formula in SAT directly corresponds to a model in MSWP (and vice versa).

3.6 3-SAT $\leq_p$ SAT

To also show the reduction from 3-SAT, we will use the intermediate reduction to SAT, the language of satisfiable boolean formulas in CNF. This relation is immediately obvious, as 3-SAT $\subset$ SAT. For the Karp reduction we have f(x) = x and x $\in$ 3-SAT if and only if f(x) $\in$ SAT.

3.7 3-SAT $\leq_p$ MSWP

Due to the transitivity of Karp reductions, we can combine the previous reductions 3-SAT $\leq_p$ SAT and SAT $\leq_p$ MSWP to conclude 3-SAT $\leq_p$ MSWP.

3.8 NP-hard & NP-complete

As shown, an arbitrary formula in 3-SAT can be converted to a world in MSWP in polynomial time. This proves that MSWP is NP-hard. Together with the fact that we have proven that MSWP is in NP, we conclude that MSWP is NP-complete.

3.9 Cost Calculation

A SAT formula inside Minesweeper consists of m number of clauses with n amount of variables.

Via the previous example we can deduce:

Bombs $m \cdot (3 \cdot 98 + (n-3) \cdot 47) + n \cdot (4+4) + m \cdot (4+1)$. Each clause has to check three literals, all other variables not in the clause are not checked, thus we have to subtract them, Input and Output Size, Clause Start + Collector

Height $4 + m \cdot 25 + 5$. Input Size, checked blocks, end size

Width $4 + n \cdot 39 + 3$. Clause Start, checked blocks, Collector Size.

Transforming a 3-SAT Problem to a Minesweeper Problem is possible and costs are P and not exponential. Thus Minesweeper is NP-complete.