On the Critical Density of Minesweeper Boards

Authors
Eytan Chong
Sim Hayden
Ma Weiyi
Brandan Goh Yu Hong

Teacher-Mentor
Mdm. Khoo Geok Hwa

Dunman High School

May 3, 2024

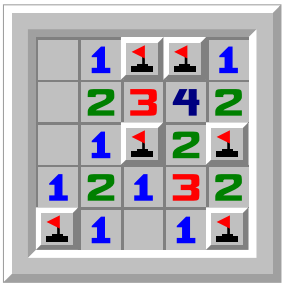
Abstract

We introduce the critical density of a Minesweeper board and investigate how the dimensions of a board affects it. We first present a mathematical analysis of Minesweeper gameplay and implement a Minesweeper solver. We then model our simulated results with the logistic function in order to calculate the critical densities of various board sizes. Lastly, we discuss the general trend of the calculated critical densities and show that the critical density converges to approximately 0.159 for all sufficiently large board sizes.

Contents

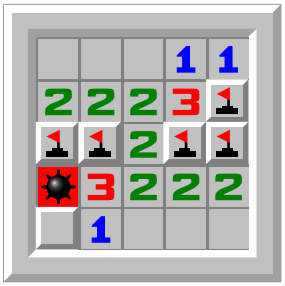
1 Introduction

Minesweeper is a single-player logic puzzle played on a grid of square cells. A number of mines are randomly distributed throughout the board. Initially, all cells are unopened. The player can open a cell by clicking it. If a mined cell is opened, the player loses. If a safe cell is opened, the number of adjacent mined cells is revealed. To win, the player must open all safe cells.



An example of a won game. All safe cells have been opened. Here, the numbers represent the number of mined cells adjacent to each cell. Note that the flags represent mined cells that have been identified by the player.

Figure 1.1



An example of a lost game. A mined cell has been opened. It is marked by the mine with a red background.

Figure 1.2

1.1 Research Problem

In this paper, we investigate the relationship between the critical density and the dimensions of a Minesweeper board. We define the critical density of a Minesweeper board to be the maximum density of mines (the ratio of the number of mines to the number of cells) that allows the player to win the game with certainty, assuming perfect play. In other words, below the critical density, a player playing perfectly can expect to win almost every game, while above the critical density, the same player can expect to lose almost every game.

1.2 Literature Review

The topic of critical densities in Minesweeper has been researched on before in the literature. Dempsey and Guinn [1] demonstrated that the critical densities of select square boards was between 0.20 and 0.30. However, the study was limited to a solver that considered only inference; no guesses were taken. Sinha et al. [2] have also observed a similar range when testing Minesweeper solvers. Percolation theory has also been used by Qing et al. [3] to determine the critical densities of select square boards, giving a more precise estimate of 0.2035 ± 0.002, though their definition of "critical density" differs from the one defined here. Nevertheless, our research

provides a more complete understanding of the critical density by accounting for guesses and boards of all sizes.

1.3 Outline

The outline of the paper is as follows. Sections 2 through 6 goes through the development of our Minesweeper solver. We begin by introducing notation and definitions for important objects in Minesweeper in Section 2. We then cover inference and guessing, the two major techniques used in solving Minesweeper, in Sections 3 and 4 respectively. In Section 5, we show that starting in the corner is optimal. Lastly, in Section 6, we consolidate the previous three sections and present an implementation of our solver.

In Section 7, we introduce the logistic function as a curve fitting model. We then define and derive a closed form for the critical density of a Minesweeper board. Next, in Section 8, we describe our methodology for finding the critical density of a given board size. In Section 9, we present and analyse our results. Finally, we conclude our paper in Section 10.

1.4 Claim of Originality

For clarity, we assert the originality of all lemmas, propositions, and algorithms presented within this paper. All mathematical constructs put forth in this work have been meticulously derived and formulated through independent research. Unless explicitly stated otherwise, the definitions provided in this paper are also fully original.

2 Preliminaries

In this section, we introduce notation for important objects in Minesweeper. We first define a Minesweeper board in Section 2.1. Then, in Section 2.2, we define other objects of interest. Throughout this section, we build upon the notation used by Crow [4] and Huang [5].

2.1 Minesweeper Board

Definition 1. A Minesweeper board B is given by the 6-tuple

$$B = (\mathcal{D}, \mathcal{A}, \mathcal{M}, \mathcal{S}, M, S) \tag{2.1}$$

where:

- $\mathcal{D}$ represents the dimensions of B,
- $\mathcal{A}$ is the set of cells on B,
- $\mathcal{M}$ and $\mathcal{S}$ represent the assigned state of B, and
- M and S represent the known state of B.

Definition 2. The dimensions $\mathcal{D}$ of a board is defined as

$$\mathcal{D} = (p, q, m) \tag{2.2}$$

where $p, q \in \mathbb{N}$ are the length and width of the board respectively and $m \in \mathbb{N}_0$ is the number of mines on the board.

In writing, it is conventional to describe a board with dimensions (p, q, m) as a "$p \times q$ / m board".

Definition 3. Given a board with dimensions $\mathcal{D} = (p, q, m)$, the set of all cells on the board is notated $\mathcal{A}$ with the definition

$$\mathcal{A} = [0, p) \times [0, q) \subset \mathbb{Z}^2 \tag{2.3}$$

Definition 4. $\mathcal{M} \subseteq \mathcal{A}$ is the set of all cells on a given board that contain mines. Elements of $\mathcal{M}$ are termed "mined cells".

Definition 5. $\mathcal{S} \subseteq \mathcal{A}$ is the set of all cells on a given board that do not contain mines. Elements of $\mathcal{S}$ are termed "safe cells".

Since $\mathcal{M}$ and $\mathcal{S}$ are a partition of $\mathcal{A}$, they obey the relation

$$\mathcal{M} = \mathcal{S}^{\complement} \tag{2.4}$$

From Definition 2, we also have

$$|\mathcal{M}| = m \tag{2.5}$$

Definition 6. $M \subseteq \mathcal{M}$ is the set of all cells on a given board that are known to be in $\mathcal{M}$.

Definition 7. $S \subseteq \mathcal{S}$ is the set of all cells on a given board that are known to be in $\mathcal{S}$.

2.2 Miscellaneous Definitions

Definition 8. $U \subseteq \mathcal{A}$ is the set of all cells on a given board whose states are unknown.

$$U = (M \cup S)^{\complement} \tag{2.6}$$

Definition 9. Given a cell $a \in \mathcal{A}$, the set of cells adjacent to a is given by the function $K \colon \mathcal{A} \to 2^{\mathcal{A}}$ with the definition

$$K(a) = \{b \in \mathcal{A} \colon D_{\text{Chebyshev}}(a, b) = 1\} \tag{2.7}$$

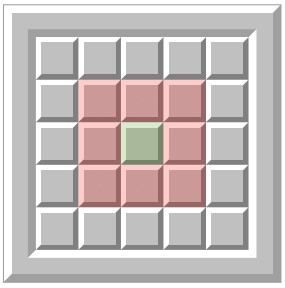


Figure 2.1

As an example, let a be the green-coloured cell in Figure 2.1. Then, $K(a)$ is the set of the red-coloured cells.

Definition 10. Given a cell $a \in \mathcal{A}$, the set of cells adjacent to a that are also in a set $X \subseteq \mathcal{A}$ is given by the function $K_X \colon \mathcal{A} \to 2^{\mathcal{A}}$ with the definition

$$K_X(a) = K(a) \cap X \tag{2.8}$$

Definition 11. The mine count (number of adjacent mined cells) of a cell $a \in S$ is given by $N \colon S \to \mathbb{Z}$ such that

$$N(a) = |K_{\mathcal{M}}(a)| \tag{2.9}$$

Definition 12. The set $\mathcal{B} \subseteq S$ is defined as

$$\mathcal{B} = \{a \in S \colon |K_U(a)| > 0\} \tag{2.10}$$

Elements of $\mathcal{B}$ are termed "boundary cells".

Definition 13. Given two boards B_1 and B_2, we say B_2 is a continuation of B_1, denoted $B_1 \Rightarrow B_2$, if the following statements hold.

$$\mathcal{D}_1 = \mathcal{D}_2 \tag{2.11}$$

$$M_1 \subseteq M_2 \tag{2.12}$$

$$S_1 \subseteq S_2 \tag{2.13}$$

Definition 14. The set of all boards is denoted $\mathbb{B}$.

3 Inference Algorithm

Inference is one of the two major techniques used in solving Minesweeper. Put simply, inference involves making deductions about the state of unknown cells based on the information available.

In this section, we discuss a novel algorithm that can infer the state of unknown cells. In Section 3.1, we show how solving Minesweeper can be reduced into a

constraint satisfaction problem. We then sketch an outline of our inference algorithm in detail in Section 3.2. Next, in Section 3.3, we discuss the individual processes that make up the inference algorithm in detail. Finally, we present an implementation of our inference algorithm in Section 3.4.

3.1 Constraint Satisfaction Problem

The main idea behind inference involves viewing the problem of solving Minesweeper into a constraint satisfaction problem [6]. We can do so by transforming all the information present on the board into a system of constraints and solving it. The most natural and common transformation is the indicator function of $\mathcal{M}$.

Definition 15. The indicator function of $\mathcal{M}$ is the function $\mathbf{1}_{\mathcal{M}} \colon \mathcal{A} \to \{0, 1\}$ such that

$$\mathbf{1}_{\mathcal{M}}(a) = \begin{cases} 0, & a \notin \mathcal{M} \\ 1, & a \in \mathcal{M} \end{cases} \tag{3.1}$$

Proposition 1. $|X \cap \mathcal{M}| = \sum_{a \in X} \mathbf{1}_{\mathcal{M}}(a)$ *for all* $X \subseteq \mathcal{A}$.

Proof.

$$\begin{aligned} \sum_{a \in X} \mathbf{1}_{\mathcal{M}}(a) &= \sum_{a \in X \cap \mathcal{M}} \mathbf{1}_{\mathcal{M}}(a) + \sum_{a \in X \cap \mathcal{S}} \mathbf{1}_{\mathcal{M}}(a) \\ &= \sum_{a \in X \cap \mathcal{M}} 1 + \sum_{a \in X \cap \mathcal{S}} 0 \\ &= |X \cap \mathcal{M}| \end{aligned} \tag{3.2}$$

□

With Proposition 1, we can easily express our Minesweeper board as a system of constraints using $\mathbf{1}_{\mathcal{M}}$. We introduce two types of constraints, namely local constraints and global constraints.

Definition 16. The local constraint of a cell $a \in S$ is defined as

$$N(a) = \sum_{b \in K(a)} \mathbf{1}_{\mathcal{M}}(b) \tag{3.3}$$

Definition 17. The global constraint of a board with m mines is defined as

$$m = \sum_{a \in \mathcal{A}} \mathbf{1}_{\mathcal{M}}(a) \tag{3.4}$$

Remark. Here, we used the fact that $N(a) = |K(a) \cap \mathcal{M}|$ and $m = |\mathcal{A} \cap \mathcal{M}|$ and applied Proposition 1.

In fact, this system of constraints is all we need for inference. By solving the

system of constraints, we can determine the state of a cell from its image under $\mathbf{1}_{\mathcal{M}}$. More concretely, for all cells $a \in \mathcal{A}$,

$$a \in \begin{cases} M, & \mathbf{1}_{\mathcal{M}}(a) = 1 \\ S, & \mathbf{1}_{\mathcal{M}}(a) = 0 \\ U, & \mathbf{1}_{\mathcal{M}}(a) \text{ unconstrained} \end{cases} \tag{3.5}$$

3.1.1 Optimised Constraints

The astute reader would have noticed that we can optimise the computation of local constraints by only considering unknown adjacent cells. More concretely, for some $a \in S$,

$$\begin{aligned} N(a) &= \sum_{b \in K(a)} \mathbf{1}_{\mathcal{M}}(b) \\ &= \sum_{b \in K_M(a)} \mathbf{1}_{\mathcal{M}}(b) + \sum_{b \in K_S(a)} \mathbf{1}_{\mathcal{M}}(b) + \sum_{b \in K_U(b)} \mathbf{1}_{\mathcal{M}}(b) \\ &= |K_M(a)| + 0 + \sum_{b \in K_U(a)} \mathbf{1}_{\mathcal{M}}(b) \end{aligned} \tag{3.6}$$

Observe now that if $a \in S \setminus \mathcal{B}$, then $|K_U(a)| = 0$, whence $\sum_{b \in K_U(a)} \mathbf{1}_{\mathcal{M}}(b) = 0$ and there is no information about unknown cells to infer. We thus only consider the cells in $\mathcal{B}$.

Definition 18. The optimised local constraint of a cell $a \in B$ is defined as

$$N(a) - |K_M(a)| = \sum_{b \in K_U(a)} \mathbf{1}_{\mathcal{M}}(b) \tag{3.7}$$

In a similar fashion, we can disregard all cells in $M \cup S$ in our computation of the global constraint since

$$\begin{aligned} m &= \sum_{a \in \mathcal{A}} \mathbf{1}_{\mathcal{M}}(a) \\ &= \sum_{a \in M} \mathbf{1}_{\mathcal{M}}(a) + \sum_{a \in S} \mathbf{1}_{\mathcal{M}}(a) + \sum_{a \in U} \mathbf{1}_{\mathcal{M}}(a) \\ &= |M| + 0 + \sum_{a \in U} \mathbf{1}_{\mathcal{M}}(a) \end{aligned} \tag{3.8}$$

Definition 19. The optimised global constraint of a board with m mines is defined as

$$m - |M| = \sum_{a \in U} \mathbf{1}_{\mathcal{M}}(a) \tag{3.9}$$

Example To illustrate the inference algorithm, consider the following $4 \times 3\,/\,3$ board in Figure 3.1.

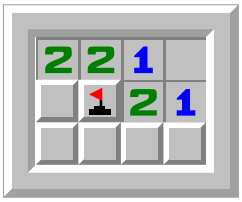


Figure 3.1

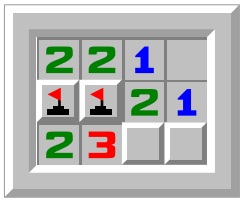


Figure 3.2

Applying Definitions 3.7 and 3.9, we have the following system of constraints. Note that we do not care about cell $(3,2)$ as it is in the set $S \setminus \mathcal{B}$.

$$\begin{cases} N((0,2)) - |K_M((0,2))| = \sum_{b \in K_U((0,2))} \mathbf{1}_{\mathcal{M}}(b) \\ N((1,2)) - |K_M((1,2))| = \sum_{b \in K_U((1,2))} \mathbf{1}_{\mathcal{M}}(b) \\ N((2,1)) - |K_M((2,1))| = \sum_{b \in K_U((2,1))} \mathbf{1}_{\mathcal{M}}(b) \\ N((3,1)) - |K_M((3,1))| = \sum_{b \in K_U((3,1))} \mathbf{1}_{\mathcal{M}}(b) \\ m - |M| = \sum_{a \in U} \mathbf{1}_{\mathcal{M}}(a) \end{cases}$$

Our system of constraints quickly simplifies to

$$\begin{cases} 1 = \mathbf{1}_{\mathcal{M}}((0,1)) \\ 1 = \mathbf{1}_{\mathcal{M}}((0,1)) \\ 1 = \mathbf{1}_{\mathcal{M}}((1,0)) + \mathbf{1}_{\mathcal{M}}((2,0)) + \mathbf{1}_{\mathcal{M}}((3,0)) \\ 1 = \mathbf{1}_{\mathcal{M}}((2,0)) + \mathbf{1}_{\mathcal{M}}((3,0)) \\ 2 = \mathbf{1}_{\mathcal{M}}((0,1)) + \mathbf{1}_{\mathcal{M}}((0,0)) + \mathbf{1}_{\mathcal{M}}((1,0)) + \mathbf{1}_{\mathcal{M}}((2,0)) + \mathbf{1}_{\mathcal{M}}((3,0)) \end{cases}$$

It is clear that $\mathbf{1}_{\mathcal{M}}((0,1)) = 1$. Substituting it back into our system yields

$$\begin{cases} 1 = \mathbf{1}_{\mathcal{M}}((1,0)) + \mathbf{1}_{\mathcal{M}}((2,0)) + \mathbf{1}_{\mathcal{M}}((3,0)) \\ 1 = \mathbf{1}_{\mathcal{M}}((2,0)) + \mathbf{1}_{\mathcal{M}}((3,0)) \\ 1 = \mathbf{1}_{\mathcal{M}}((0,0)) + \mathbf{1}_{\mathcal{M}}((1,0)) + \mathbf{1}_{\mathcal{M}}((2,0)) + \mathbf{1}_{\mathcal{M}}((3,0)) \end{cases}$$

Subtracting the second equation from the first equation gives us the constraint

$$\mathbf{1}_{\mathcal{M}}((1,0)) = 0$$

Similarly, subtracting the first equation from the third equation gives

$$\mathbf{1}_{\mathcal{M}}((0,0)) = 0$$

Substituting these values back into the system leaves us with the single constraint

$$1 = \mathbf{1}_{\mathcal{M}}((2,0)) + \mathbf{1}_{\mathcal{M}}((3,0))$$

Observe now that both $\mathbf{1}_{\mathcal{M}}((2,0)) = 0$ and $\mathbf{1}_{\mathcal{M}}((2,0)) = 1$ yield valid solutions to the above constraint. Similarly, both $\mathbf{1}_{\mathcal{M}}((3,0)) = 0$ and $\mathbf{1}_{\mathcal{M}}((3,0)) = 1$ yield valid solutions. We thus have that $\mathbf{1}_{\mathcal{M}}((2,0))$ and $\mathbf{1}_{\mathcal{M}}((3,0))$ are both unconstrained.

To sum, we have inferred the following assignments.

$$\begin{cases} \mathbf{1}_{\mathcal{M}}((0,1)) = 1 \\ \mathbf{1}_{\mathcal{M}}((0,0)) = 0 \\ \mathbf{1}_{\mathcal{M}}((1,0)) = 0 \\ \mathbf{1}_{\mathcal{M}}((2,0)) \text{ unconstrained} \\ \mathbf{1}_{\mathcal{M}}((3,0)) \text{ unconstrained} \end{cases}$$

By Relation 3.5, we conclude that $(0,1) \in M$ and $(0,0), (1,0) \in S$, while $(2,0)$ and $(3,0)$ remain in U. This finally gives us the board depicted in Figure 3.2.

3.2 Sketch

In order to discuss our implementation of the inference algorithm, we must first understand the processes and decisions we made in the example above that allowed us to solve the board.

We first converted the given board state into a system of constraints using Formulas 3.7 and 3.9. Then, we observed that there was a trivial solution to a constraint, namely $\mathbf{1}_{\mathcal{M}}((0,1)) = 1$, and substituted it back into the system of constraints. As there were no more trivial solutions, we began constructing new constraints by subtracting existing constraints from one another. This led to the creation of two new trivial solutions ($\mathbf{1}_{\mathcal{M}}((0,0)) = 0$ and $\mathbf{1}_{\mathcal{M}}((1,0)) = 0$). Once again, we substituted them back into the system. Finally, we were left with a single constraint. With no more trivial solutions and no new constraints to construct, we could not infer any further and thus halted.

Indeed, this simple example is enough to provide an intuition for our inference algorithm. The flow of the algorithm is illustrated in Figure 3.3.

3.3 Processes

In this section, we discuss the details of the four main processes of our inference algorithm. As listed in Figure 3.3, they are:

1. Deriving constraints (Section 3.3.1)
2. Solving trivial constraints (Section 3.3.2)
3. Constructing constraints (Section 3.3.3)
4. Assigning solved cells (Section 3.3.4)

To aid our discussion of the various processes, we first introduce the following definitions.

Definition 20. A constraint C is defined by the tuple

$$C = (s, X) \tag{3.10}$$

where $s \in \mathbb{N}_0$ is said to be the sum of C and $X \subseteq \mathcal{A}$ is said to be the cells of C.

More familiarly, a constraint (s, X) can be expressed as $s = \sum_{a \in X} \mathbf{1}_{\mathcal{M}}(a)$.

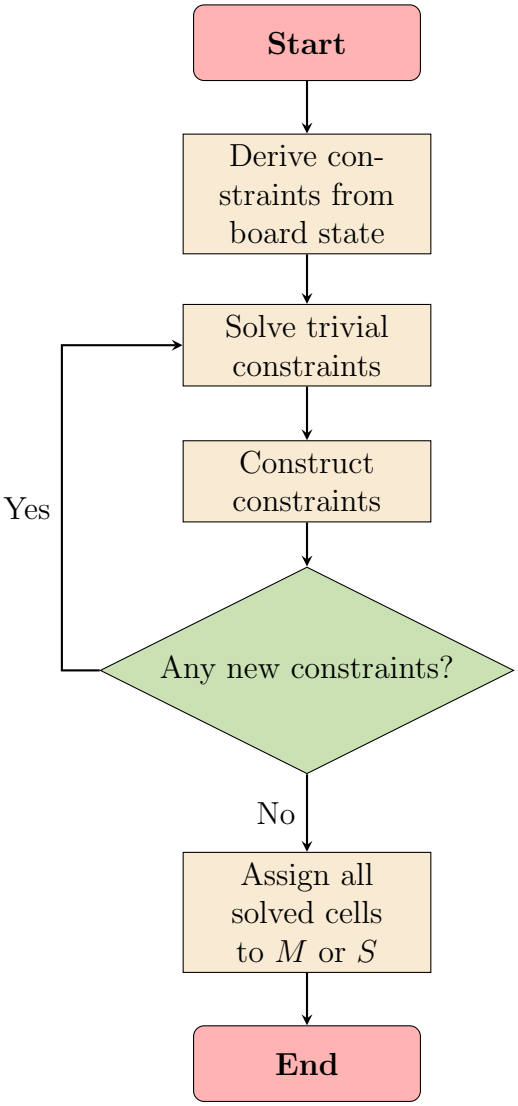


Figure 3.3
A flowchart of our inference algorithm.

Definition 21. The set of all true constraints (within the context of the same board) is denoted $\mathcal{C}$.

Definition 22. A constraint $(s, X) \in \mathcal{C}$ is said to be solved if

$$|X| = 1 \tag{3.11}$$

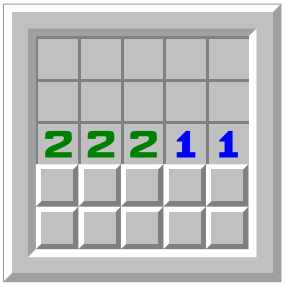


Figure 3.4

Consider the $5 \times 5\,/\,5$ board on the left. The constraint $(2, \{(0,1),(1,1)\})$ is in $\mathcal{C}$ since it is true. The constraint $(2, \{(3,1),(4,1)\})$ is not in $\mathcal{C}$ since it is false.

3.3.1 Deriving Constraints

This process entails the derivation of constraints from a given board state. As previously discussed in Section 3.1, we can do so by constructing the optimised local constraints of all boundary cells and the board's optmised global constraint.

Algorithm 3.1 Derives constraints from the given board state

Input: $B \in \mathbb{B}$
Output: `constraints` $\subseteq \mathcal{C}$

```
1: procedure DeriveConstraints(B)
2:     constraints ← ∅

       // Add optimised local constraint for each boundary cells (see Definition 3.7)
3:     for all a ∈ ℬ do
4:         constraints ← constraints ∪ (N(a) − |K_M(a)|, K_U(a))
5:     end for

       // Add optimised global constraint (see Definition 3.9)
6:     constraints ← constraints ∪ (m − |M|, U)

7:     return constraints
8: end procedure
```

3.3.2 Solving Trivial Constraints

As the name suggests, this process solves constraints that have an obvious solution.

In order to define what makes a solution obvious, consider the constraint $C = (s, X)$, where $X \subseteq U$. There are exactly $\binom{|X|}{s}$ ways that C can be solved. Observe that $\binom{|X|}{s} = 1$ if and only if s is 0 or $|X|$, whence C has a unique solution and is thus considered trivial.

Definition 23. A constraint $(s, X) \in \mathcal{C}$ is said to be trivial if one of the following two conditions hold:

$$s = 0 \tag{3.12}$$

$$s = |X| \tag{3.13}$$

Lemma 1. *For finite sets A and B, if $A \cap B = \varnothing$, then $|A| + |B| = |A \cup B|$.*

Proof. The inclusion-exlusion principle states $|A \cup B| = |A| + |B| - |A \cap B|$. Since $A \cap B = \varnothing$, it follows that $|A \cap B| = 0$. Our desired result follows immediately. □

Lemma 2. *For finite sets A and B, if $A \subseteq B$ and $|A| = |B|$, then $A = B$.*

Proof. Let $C = B \setminus A$. Then $A \cup C = B$, whence $|A \cup C| = |B|$. By Lemma 1, $|A| + |C| = |B|$, hence $|C| = 0$. Thus, $C = \varnothing$ and $A = B$. □

Proposition 2. $(0, X) \in \mathcal{C} \implies \forall\, a \in X \colon (0, a) \in \mathcal{C}$

Proof. By Proposition 1, it suffices to show that $|X \cap \mathcal{M}| = 0$ implies $X \subseteq \mathcal{S}$. Since $|X \cap \mathcal{M}| = 0 = |\varnothing|$ and $\varnothing \subseteq X \cap \mathcal{M}$, by Lemma 2, we have that $X \cap \mathcal{M} = \varnothing$. Thus, $X \subseteq \mathcal{M}^{\complement} = \mathcal{S}$. □

Proposition 3. $(|X|, X) \in \mathcal{C} \implies \forall\, a \in X \colon (1, a) \in \mathcal{C}$

Proof. By Proposition 1, it suffices to show that $|X \cap \mathcal{M}| = |X|$ implies $X \subseteq \mathcal{M}$. Observe that $X \cap \mathcal{M} \subseteq X$. By Lemma 2, $X \cap \mathcal{M} = X$, thus $X \subseteq \mathcal{M}$. □

Algorithm 3.2 Solves trivial constraints

Input: `constraints` $\subseteq \mathcal{C}$
Output: `solvedConstraints` $\subseteq \mathcal{C}$

```
1: procedure SolveTrivials(constraints)
2:     solvedConstraints ← ∅

3:     for all (s, X) ∈ constraints do
           // Apply Proposition 2
4:         if s = 0 then
5:             for all a ∈ X do
6:                 solvedConstraints ← solvedConstraints ∪ (0, {a})
7:             end for

           // Apply Proposition 3
8:         else if s = |X| then
```

```
            for all a ∈ X do
                solvedConstraints ← solvedConstraints ∪ (1, {a})
            end for
        end if
    end for

    return solvedConstraints
end procedure
```

3.3.3 Constructing Constraints

As previously discussed, we can construct new constraints by subtracting existing ones from one another. As an example, consider the constraints $C_1 = (2, \{a, b, c\})$ and $C_2 = (1, \{a, b\})$. We can subtract C_2 from C_1 to construct the constraint $(2 - 1, \{a, b, c\} \setminus \{a, b\}) = (1, \{c\})$.

It must be noted, however, that not all pairs of constraints can be subtracted; for a pair of constraints C_1 and C_2 with cells X_1 and X_2 respectively, we require $X_2 \subset X_1$ for C_2 to be subtracted from C_1 so as to ensure that the resultant sum $\sum_{a \in X_2 \setminus X_1} \mathbf{1}_{\mathcal{M}}(a)$ is well defined.

Definition 24. Given two constraints $C_1 = (s_1, X_1) \in \mathcal{C}$ and $C_2 = (s_2, X_2) \in \mathcal{C}$ such that $X_1 \supset X_2$, the difference $C_1 - C_2$ is defined as

$$C_1 - C_2 = (s_1 - s_2, X_1 \setminus X_2) \tag{3.14}$$

Algorithm 3.3 Constructs constraints

Input: `constraints` $\subseteq \mathcal{C}$
Output: `constructed` $\subseteq \mathcal{C}$

```
 1: procedure ConstructConstraints(constraints)
 2:     constructed ← ∅

        // Iterate through all pairs of constraints
 3:     for all (s1, X1) ∈ constraints do
 4:         for all (s2, X2) ∈ constraints do
                // Check that subtraction is well defined
 5:             if X1 ⊃ X2 then
                    // Subtract the two constraints (see Definition 24)
 6:                 constructed ← constructed ∪ (s1 − s2, X1 \ X2)
 7:             end if
 8:         end for
 9:     end for

10:     return constructed
11: end procedure
```

3.3.4 Assigning Solved Cells

Lastly, we assign the cells of solved constraints to either M or S by checking the sum of each solved constraint and applying Relation 3.5.

Algorithm 3.4 Assigns solved cells to M or S

Input: `solvedConstraints` $\subseteq \mathcal{C}$
Output: $M' \supseteq M$, $S' \supseteq S$

```
 1: procedure AssignSolvedCells(solvedConstraints)
 2:     M' ← M
 3:     S' ← S
 4:     for all (s, X) ∈ constraints do
            // Check that the constraint is solved
 5:         if |X| = 1 then
                // Cell is safe
 6:             if s = 0 then
 7:                 S' ← S' ∪ X

                // Cell is mined
 8:             else if s = 1 then
 9:                 M' ← M' ∪ X
10:             end if
11:         end if
12:     end for

13:     return M', S'
14: end procedure
```

3.4 Implementation

Our implementation of the inference algorithm, written in pseudocode, is as follows. The source code can be found in Annex A.1.

Algorithm 3.5 Inference algorithm

Input: $B \in \mathbb{B}$
Output: $B' \in \mathbb{B} \colon B \Rightarrow B'$

```
 1: procedure Infer(B)
 2:     B' ← B
 3:     newConstraintsConstructed ← true
 4:     oldConstraints ← ∅

        // Derive constraints from board state (see Algorithm 3.1)
 5:     constraints ← DeriveConstraints(B)

        // Keep inferring until no new constraints can be constructed
 6:     while newConstraintsConstructed do
            // Solve trivial constraints (see Algorithm 3.2)
 7:         constraints ← constraints ∪ SolveTrivials(constraints)
```

```
        // Construct constraints (see Algorithm 3.3)
8:      constructed ← ConstructConstraints(constraints)
9:      constraints ← constraints ∪ constructed

        // Determine if any new constraints have been constructed
10:     if oldConstraints = constraints then
11:         newConstraintsConstructed ← false
12:     end if
13:     oldConstraints ← constraints
14:   end while

      // Assign solved cells to M or S (see Algorithm 3.4)
15:   M′, S′ ← AssignSolvedCells(solvedConstraints)

16:   return B′
17: end procedure
```

4 Guessing Algorithm

Guessing is the second major technique used in solving Minesweeper. As the name suggests, guessing involves making educated guesses on the state of cells.

In this section, we discuss an original algorithm that determines the best cell to be opened. In Section 4.1 we introduce the notion of safety as a metric to base our guessing algorithm on. We then sketch an outline of our guessing algorithm in Section 4.2. Lastly, in Section 4.3, we discuss our implementation of the guessing algorithm while going through its individual processes in detail.

To aid our discussion, we introduce the following notation.

Definition 25. The set of all boards that continue a board B is denoted $\overset{\Rightarrow}{B}$.

Definition 26. The set $E \subseteq U$ is defined as

$$E = \{a \in U : |K_{\mathcal{B}}(a)| > 0\} \tag{4.1}$$

Elements of E are termed "exposed cells".

Definition 27. The set $F \subseteq U$ is defined as

$$F = \{a \in U : |K_{\mathcal{B}}(a)| = 0\} \tag{4.2}$$

Elements of F are termed "floating cells".

4.1 Safety

Safety is the metric that we will be considering in our guessing algorithm. The safety of a cell is defined as the probability that it is safe.

Definition 28. The safety of a cell $a \in U$ is denoted $P_S(a)$ and is defined as

$$P_S(a) = \mathbb{P}(a \in \mathcal{S}) \tag{4.3}$$

4.1.1 Configurations and Weights

It is obvious that the safety of a cell $a \in U$ on a board B can be calculated as

$$P_S(a) = \frac{|\{B' \in \overrightarrow{B} : a \in \mathcal{S}'\}|}{|\overrightarrow{B}|} \tag{4.4}$$

To calculate $P_S(a)$, one could naïvely generate all boards in $\overrightarrow{B}$ by brute force. While this could work for smaller boards, it is unsustainable in the long run. To give a sense of perspective, there are more $20 \times 20 \,/\, 75$ boards than atoms in the universe.

To circumvent this problem, we introduce the concept of configurations and weights. Put simply, a configuration C of a board B is a unique way to "assign" all cells in E into either $\mathcal{M}$ or $\mathcal{S}$. The corresponding weight W_C is the number of unique ways to "assign" the remaining mines into the remaining unknown cells, i.e. F. We can hence calculate $|\overrightarrow{B}|$ as the sum of the weights of all configurations.

$$|\overrightarrow{B}| = \sum_{C} W_C \tag{4.5}$$

We go through the calculation of $|\{B' \in \overrightarrow{B} : a \in \mathcal{S}'\}|$ in the subsequent sections. For now, we rigorously define and introduce notion for configurations and weights.

Definition 29. A configuration C of a board B is defined by the 3-tuple

$$C = (C_M, C_S, C_U) \tag{4.6}$$

such that

- C_M, C_S and C_U are a partition of E
- there exists a board $B' \in \overrightarrow{B}$ such that $C_M \subseteq \mathcal{M}'$ and $C_S \subseteq \mathcal{S}'$.

Definition 30. A configuration C is said to be proper if $C_U = \varnothing$.

Definition 31. The set of all configurations of a board B is denoted $\mathrm{Con}(B)$.

Definition 32. The set of all proper configurations of a board B is denoted $\mathrm{Con}_P(B)$.

Definition 33. Given a proper configuration $C \in \mathrm{Con}_P(B)$, its weight W_C is defined as

$$W_C = |\{B' \in \overrightarrow{B} : C_M \subseteq \mathcal{M}' \wedge C_S \subseteq \mathcal{S}\}| \tag{4.7}$$

Proposition 4. *Given a proper configuration $C \in \mathrm{Con}_P(B)$, its corresponding weight $W_C = \binom{|F|}{m-|M|-|C_M|}$*

Proof. Observe that there are $|F|$ remaining unknown cells and $m-|M|-|C_M|$ mines left to "assign". W_C is thus equivalent to the number of ways to choose $m - |M| - |C_M|$ objects (mines) from $|F|$ total objects (unknown cells). □

With our definitions, we can refine Equation 4.5 as follows

$$|\overrightarrow{B}| = \sum_{C \in \mathrm{Con}_P(B)} W_C \tag{4.8}$$

4.1.2 Exposed Cells

In this section, we calculate $P_S(a)$ for some $a \in E$. From Definition 29, we see that $a \in \mathcal{S}'$ for some board $B' \in \overrightarrow{B}$ if and only if $a \in C_S$ for some proper configuration $C \in \mathrm{Con}_P(B)$. Hence,

$$|\{B' \in \overrightarrow{B} : a \in \mathcal{S}'\}| = \sum_{\substack{C \in \mathrm{Con}_P(B) \\ a \in S_C}} W_C \tag{4.9}$$

Thus, from Equations 4.4 and 4.8,

$$P_S(a) = \sum_{\substack{C \in \mathrm{Con}_P(B) \\ a \in S_C}} W_C \Bigg/ \sum_{C \in \mathrm{Con}_P(B)} W_C \tag{4.10}$$

4.1.3 Floating Cells

In this section, we calculate $P_S(a)$ for some $a \in F$ on a board B. We begin by proving that the expected number of mines in E is equal to $\sum_{a \in E} \mathbb{P}(a \in \mathcal{M})$.

Proposition 5. $\mathbb{E}[|E \cap \mathcal{M}|] = \sum_{a \in E} \mathbb{P}(a \in \mathcal{M})$

Proof. Since $\mathcal{M} = \mathcal{S}^{\complement}$, it suffices to show that

$$\mathbb{E}[|E \cap \mathcal{S}|] = \sum_{a \in E} \mathbb{P}(a \in \mathcal{S}) \tag{4.11}$$

Consider the LHS. From the definition of an expected value,

$$\mathbb{E}[|E \cap \mathcal{S}|] = \sum_{n=0}^{|E|} n \cdot \frac{|\{B' \in \overrightarrow{B} : |E \cap \mathcal{S}'| = n\}|}{|\overrightarrow{B}|} \tag{4.12}$$

Now consider the RHS. Since $\mathbb{P}(a \in \mathcal{S}) = P_S(a)$, from Equation 4.4,

$$\sum_{a\in E}\mathbb{P}(a\in\mathcal{S})=\sum_{a\in E}\frac{|\{B'\in\overrightarrow{B}\colon a\in\mathcal{S}'\}|}{|\overrightarrow{B}|} \tag{4.13}$$

It thus suffices to show

$$\sum_{n=0}^{|E|}n\cdot|\{B'\in\overrightarrow{B}\colon|E\cap\mathcal{S}'|=n\}|=\sum_{a\in E}|\{B'\in\overrightarrow{B}\colon a\in\mathcal{S}'\}| \tag{4.14}$$

Finally, consider a proper configuration $C\in\mathrm{Con}_P(B)$ with $|E\cap\mathcal{S}_C|=n_C$. Observe that $C\in\{B'\in\overrightarrow{B}\colon|E\cap\mathcal{S}'|=n_C\}$. Thus, C contributes n_C to the sum on the LHS. Furthermore, since $|E\cap\mathcal{S}_C|=n_C$, it follows that C also contributes n_C to the sum on the RHS. Hence, after considering all configurations in $\mathrm{Con}(B)$, we see that Equality 4.14 must hold and thus our desired result is true. □

We now relate $\mathbb{E}[|E\cap\mathcal{M}|]$ and $\mathbb{E}[|F\cap\mathcal{M}|]$.

Proposition 6. $\mathbb{E}[|F\cap\mathcal{M}|]=m-|M|-\mathbb{E}[|E\cap\mathcal{M}|]$

Proof. Observe that $\mathcal{A}=M\cup S\cup E\cup F$. Hence,

$$\mathcal{A}\cap\mathcal{M}=\bigcup_{X\in\{M,S,E,F\}}X\cap\mathcal{M} \tag{4.15}$$

Since M, S, E and F are all pairwise disjoint,

$$|\mathcal{A}\cap\mathcal{M}|=\sum_{X\in\{M,S,E,F\}}|X\cap\mathcal{M}| \tag{4.16}$$

Firstly, we clearly have $|\mathcal{A}\cap\mathcal{M}|=m$. Secondly, since $M\subseteq\mathcal{M}$, we have $|\mathcal{A}\cap\mathcal{M}|=|M|$. Lastly, $S\cap\mathcal{M}=\varnothing$, hence $|S\cap\mathcal{M}|=0$. Simplifying our sum and rearranging some terms, we obtain

$$|F\cap\mathcal{M}|=m-|M|-|E\cap\mathcal{M}| \tag{4.17}$$

It quickly follows that

$$\mathbb{E}[|F\cap\mathcal{M}|]=m-|M|-\mathbb{E}[|E\cap\mathcal{M}|] \tag{4.18}$$

□

In a manner similar to that of Proposition 4, we can calculate the expected number of boards in $\overrightarrow{B}$ as

$$\mathbb{E}[|\overrightarrow{B}|]=\binom{|F|}{\mathbb{E}[|F\cap\mathcal{M}|]} \tag{4.19}$$

Likewise, we can calculate the expected number of boards in $\overrightarrow{B}$ where some $a\in F$ is safe as

$$\mathbb{E}[|\{B' \in \overset{\Rightarrow}{B} \colon a \in \mathcal{S}'\}|] = \binom{|F|-1}{\mathbb{E}[|F \cap \mathcal{M}|]} \tag{4.20}$$

From Equation 4.4,

$$P_S(a) = \binom{|F|-1}{\mathbb{E}[|F \cap \mathcal{M}|]} \Big/ \binom{|F|}{\mathbb{E}[|F \cap \mathcal{M}|]} \tag{4.21}$$

Using the definition of a binomial coefficient and simplifying, we thus obtain the following formula for the safety of a floating cell.

$$P_S(a) = 1 - \frac{\mathbb{E}[|F \cap \mathcal{M}|]}{|F|} \tag{4.22}$$

As a final remark, we note that $P_S(a)$ is independent of a. Hence, the safety of all floating cells are equal.

4.2 Sketch

In this section, we sketch an outline of our guessing algorithm. In order to develop an intuition for the flow of our guessing algorithm, we consider the $5 \times 5\,/\,6$ board B in Figure 4.1.

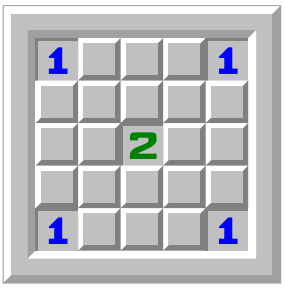


Figure 4.1

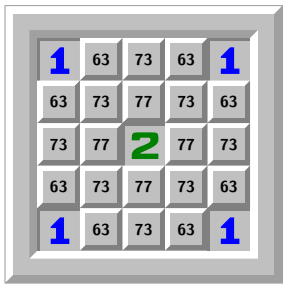


Figure 4.2

After finding all proper configurations in $\mathrm{Con}_P(B)$ and calculating their corresponding weights, one can easily calculate the safety of all unknown cells with Formulas 4.10 and 4.22. Figure 4.2 shows the safety of each unknown cell, scaled by 100. Finally, we guess the cell that has the highest safety. In our example, the cells $(1,2)$, $(2,1)$, $(2,4)$ and $(3,2)$ all have the highest score. We thus select one of them at random to guess. Figure 4.3 illustrates the flow of our guessing algorithm.

4.3 Processes

In this section, we discuss the details of the three main processes of our guessing algorithm. As listed in Figure 4.3, they are:

1. Determining configurations (Section 4.3.1)
2. Calculating safety (Section 4.3.2)
3. Guess cell with highest safety (Section 4.3.3)

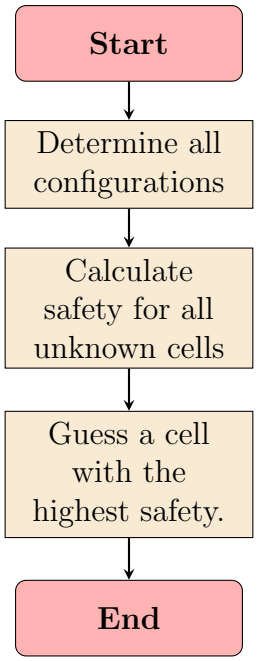


Figure 4.3
A flowchart of our guessing algorithm.

4.3.1 Determining Configurations

In order to determine all proper configurations of a given board B, we will be employing a recursive algorithm. Consider an improper configuration C and some cell $a \in C_U$. Then, we run the inference algorithm on B, with the assumptions that $a \in \mathcal{S}$, $C_S \subseteq \mathcal{S}$ and $C_M \subseteq \mathcal{M}$. The inference algorithm then returns more solved constraints, which we use to generate a new configuration C'. If C' is proper, we add it to our list of proper configurations. If not, we run the entire algorithm once more on C' and add the results to our list. Next, we do the same procedure described above, this time with the initial assumption that $a \in \mathcal{M}$. Lastly, we return our list of proper configurations. A sketch is provided in Figure 4.4.

4.3.2 Calculating Safety

As previously discussed in Section 4.1, the safety of exposed cells can be calculated using Formula 4.10, while the safety of floating cells can be calculated using Formula 4.22.

Algorithm 4.1 Calculates safety of all unknown cells

Input: $B \in \mathbb{B}$
Output: `safety` $\subseteq \mathcal{A} \times \mathbb{R}$

1: **procedure** CalculateSafety(B)
2: `configurations` $\leftarrow$ GetConfigurations($(\varnothing, \varnothing, E)$) (see Figure 4.4)
3: `safety` $\leftarrow \varnothing$

// Get combined weight of all configurations
4: `totalWeight` $\leftarrow 0$
5: **for all** $C \in$ `configurations` **do**

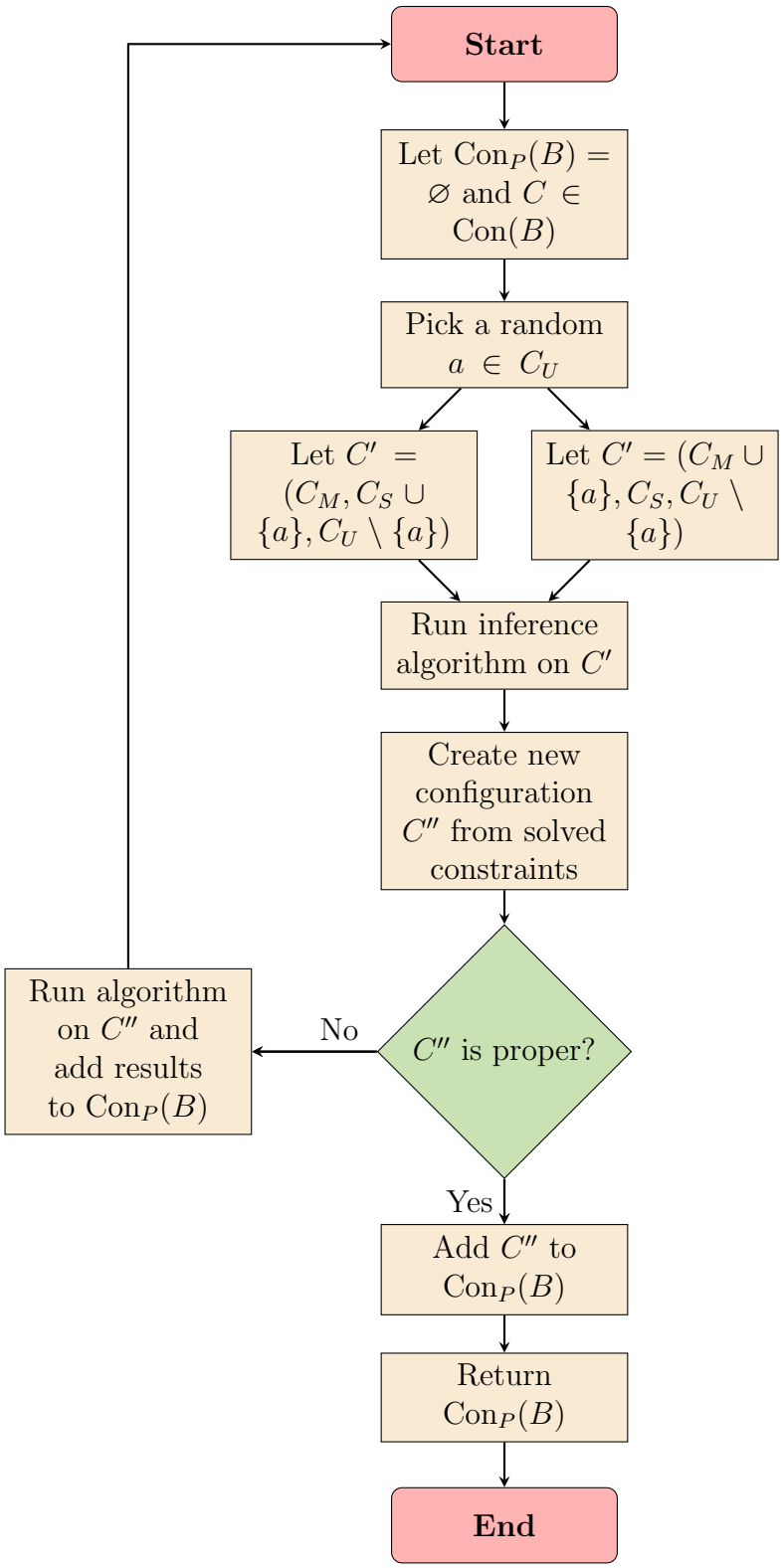


Figure 4.4
A flowchart of the GetConfigurations *algorithm.*

```
6:      end for
7:      totalWeight ← totalWeight + W_C

        // Calculate safety of exposed cells (see Formula 4.1.2)
8:      expectedExposedMines ← |E|
9:      for all exposedCell ∈ E do
10:         safeConfigurations ← 0
11:         for all C ∈ configurations do
12:             if exposedCell ∈ C_S then
13:                 safeConfigurations ← safeConfigurations + W_C
14:             end if
15:         end for
16:         exposedSafety ← safeConfigurations/totalWeight
17:         expectedExposedMines ← expectedExposedMines − exposedSafety
18:         safety ← safety ∪ (exposedCell, exposedSafety)
19:     end for

        // Calculate safety of floating cells (see Formula 4.22)
20:     expectedFloatingMines ← m − |M| − expectedExposedMines
21:     for all floatingCell ∈ F do
22:         safety ← safety ∪ (floatingCell, 1 − expectedFloatingMines/|F|)
23:     end for

24:     return safety
25: end procedure
```

4.3.3 Guess Cell with Highest Safety

Lastly, we find all cells that have the highest safety and guess one of them at random. The following algorithm describes this process.

Algorithm 4.2 Guesses a random cell with the highest safety

Input: $B \in \mathbb{B}$

```
1: procedure GUESS(B)
       // Get safety of all unknown cells
2:     allSafety ← CALCULATESAFETY(B) (see Algorithm 4.1)

       // Gets highest safety value
3:     highestSafety ← max {safety : (cell, safety) ∈ allSafety}

       // Guess a cell with the highest safety at random
4:     for all (cell, safety) ∈ allSafety do
5:         if safety = highestSafety then
6:             return cell
7:         end if
8:     end for
9: end procedure
```

5 First Click

In most modern implementations of Minesweeper, the first click, despite being a guess, is guaranteed to be safe. In this section, we show that starting at a corner is optimal. We do so by showing that $\mathbb{P}(N(a) = 0)$ for some cell a decreases with respect to $|K(a)|$.

Lemma 3. *If $U = \mathcal{A}$, then $\mathbb{P}(N(a) = 0) \propto \binom{pq-|K(a)|-1}{m}$ for all $a \in \mathcal{A}$.*

Proof. Consider some cell $a \in \mathcal{A}$. In order for $N(a)$ to be 0, we must have $K(a) \cup a \subseteq \mathcal{S}$. We must thus assign m mines in the remaining $|\mathcal{A} \setminus (K(a) \cup a)| = pq - |K(a) - 1$ cells. There are hence $\binom{pq-|K(a)|-1}{m}$ total possible continuations of B where $N(a) = 0$. Since the total number of continuations of B is constant, we have our desired result. □

Proposition 7. *For distinct cells $a, b \in F$, if $|K(a)| > |K(b)|$, then $\mathbb{P}(N(a) = 0) \leq \mathbb{P}(N(b) = 0)$.*

Proof. Since $|K(a)| > |K(b)|$, we have $pq - |K(a)| - 1 - m + k < pq - |K(b)| - 1 - m + k$ for all $k \in \mathbb{Z}$. We thus obtain the inequality

$$\prod_{k=1}^{m}(pq - |K(a)| - 1 - m + k) \leq \prod_{k=1}^{m}(pq - |K(b)| - 1 - m + k) \tag{5.1}$$

Observe that this is equivalent to

$$\frac{(pq - |K(a)| - 1)!}{(pq - |K(a)| - 1 - m)!} \leq \frac{(pq - |K(b)| - 1)!}{(pq - |K(b)| - 1 - m)!} \tag{5.2}$$

By the definition of a binomial coefficient, we have

$$\binom{pq - |K(a)| - 1}{m} \leq \binom{pq - |K(b)| - 1}{m} \tag{5.3}$$

Finally, an application of Lemma 3 gives $\mathbb{P}(N(a) = 0) \leq \mathbb{P}(N(b) = 0)$. □

Since corner, edge and centre cells each have 3, 5 and 8 adjacent cells respectively, we conclude that starting the game in a corner is optimal. To simplify matters, we always start by opening $(0, 0)$.

6 Minesweeper Solver

Now that we have discussed both our inference and guessing algorithms, we can finally piece them together to complete our original Minesweeper solver.

Intuitively, to maximise our win rate, we should never guess if we can infer. This simple rule is the basis of our solver; we solve a board by repeatedly running our inference algorithm until no inferences can be made, in which case we run our guessing algorithm. We then repeat this process until the game ends.

A sketch of our Minesweeper solver is illustrated in Figure 6.1. The relevant source code can be found in Annex A.1.

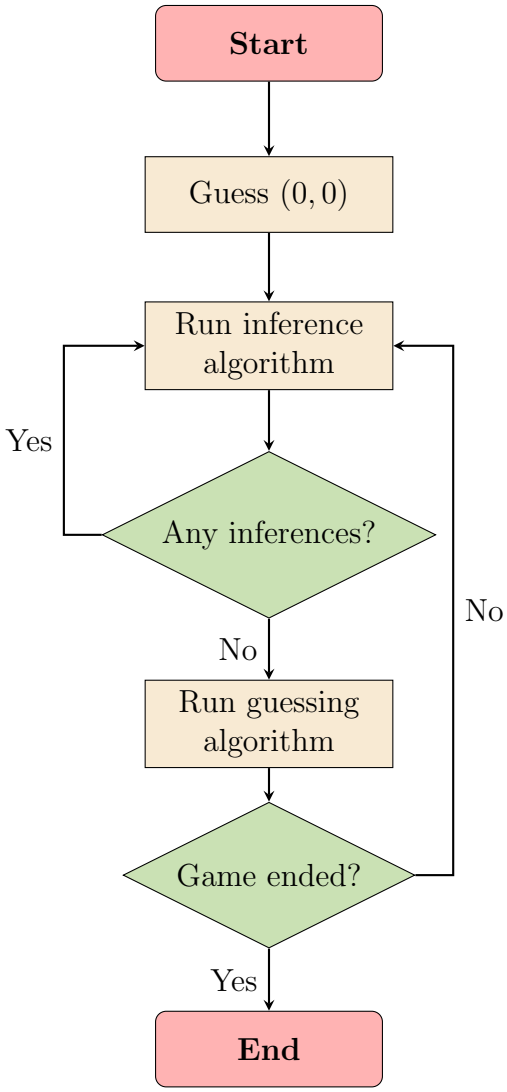


Figure 6.1
A flowchart of our solver.

7 Critical Density

Recall in the statement of our research problem that we defined the critical density of Minesweeper board to be the maximum density of mines that allows the player to win the game with certainty, assuming perfect play. Now that we have a Minesweeper solver that allows us to play almost perfectly, we aim to establish a closed form for the critical density of a board.

Definition 34. The mine density ρ of a Minesweeper board with dimensions (p, q, m) is given by

$$\rho = \frac{m}{pq} \tag{7.1}$$

Definition 35. The win rate of a Minesweeper board is denoted W.

We first consider a board B with dimensions (p, q, m). Then, we plot W as ρ varies from 0 to 1. It is obvious that for low mine densities, W will be quite high. Likewise, for high mine densities, W will be quite low. This motivates us to model the aforementioned curve with the logistic function.

Definition 36. The logistic function $L\colon [0,1] \to \mathbb{R}$ is defined as

$$L(\rho) = \frac{L}{1 + e^{k(\rho - Q)}} + b \tag{7.2}$$

for some constants $L, k, Q, b \in \mathbb{R}$ such that $L, k > 0$.

Since the critical density P is indicated by a sharp drop in the plot of W against ρ, we define P to be the mine density at which the minimum of $L'(\rho)$ is achieved.

Definition 37. The critical density P of a Minesweeper board is defined as

$$P = \arg\min_{\rho} L'(\rho) \tag{7.3}$$

We now derive a closed form for P. We first show that the only zero of $L''(\rho)$ is Q.

Proposition 8. $L''(\rho) = 0 \iff \rho = Q$.

Proof. From Definition 36, one can easily show that

$$L''(\rho) = L^2 k^2 e^{k(\rho - Q)} \cdot \frac{e^{k(\rho - Q)} - 1}{(1 + e^{k(\rho - Q)})^3} \tag{7.4}$$

It is hence obvious that ($\Longleftarrow$) is true. We now consider ($\Longrightarrow$). Note that $L, k, e^{k(\rho - Q)} > 0$. Thus, for $L''(\rho)$ to be 0,

$$e^{k(\rho - Q)} - 1 = 0 \tag{7.5}$$

whence ρ can only be Q. □

Since Q is the only zero of $L'\prime(\rho)$ and $L'(\rho) \leq 0$ for all ρ, it must be that Q the minimum of $L'(\rho)$ is achieved when $\rho = Q$. We can thus conclude that $P = Q$ when $0 \leq Q \leq 1$. If $Q < 0$, we take $P = 0$. Likewise, if $Q > 1$, we take $P = 1$. We thus have the following closed form for P.

$$P = \begin{cases} 0, & Q < 0 \\ Q, & 0 \leq Q \leq 1 \\ 1, & Q > 1 \end{cases} \tag{7.6}$$

8 Methodology

In this section, we go through our methodology of determining the critical density P of a Minesweeper board of size p by q.

8.1 W-correction

Before we discuss our methodology, we must note that the plot of W against ρ does not exactly fit a logistic curve. In particular, as ρ tends towards 1, W suddenly shoots up to 1. Consider a 4×4 / 15 board as an example. Despite having a high mine density of 0.9375, this board has a 100% win rate because the first click is guaranteed to be safe. Likewise, the win rate of a 4×4 / 14 board is approximately 0.07. This unexpected trend can be attributed to sheer luck; after revealing a guaranteed safe cell on the first click, the only realistic way of winning is to correctly guess the remaining $pq - m - 1$ safe cells at random. This results in an approximate win rate of $\prod_{k=1}^{pq-m-1} \frac{pq-m-1-k}{pq-k}$. To counter this, we implement the following two strategies.

Firstly, if the win rate of a $p \times q$ / m board has been simulated to be 0, then we take the win rate of a $p \times q$ / m' board for all $m' > m$ to also be 0. This strategy works well against larger boards; it is unlikely for the win rate of a large board to never hit 0. However, this strategy fails against smaller boards as there are not enough possible values of m to allow W to fall to 0 before shooting back up to 1. The second strategy addresses this shortcoming.

Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two board dimensions such that $m_1 < m_2$. If $\rho_1 > 0.75$ and $W_1 < W_2$, we immediately take W_2 to be 0. This effectively stops the above phenomenon before it can happen.

Altogether, we call these two strategies "W-correction".

8.2 Sketch

We now sketch an outline of our methodology. We begin by generating 1000 Minesweeper boards with dimensions (p, q, m) for all $0 < m < pq$. We then solve the generated boards using our Minesweeper solver as described in Section 6. We then record our solver's win rate for each m, enforcing W-correction in the process. Next, we plot W against ρ. We then use Powell's dog-leg algorithm [7], a popular optimisation algorithm, to fit the logistic curve $L(\rho)$ to the resulting plot. This returns the values of the parameters of $L(\rho)$. Lastly, as per Equation 7.6, we can easily calculate P from Q.

9 Results and Analysis

Using our methodology, we calculate the critical densities of all $p \times q$ Minesweeper boards where $p, q \leq 10$ and plot them in Figure 9.1. In Figure 9.2, we plot P against the aspect ratio $\frac{p}{q}$ as q runs from 2 to 10.

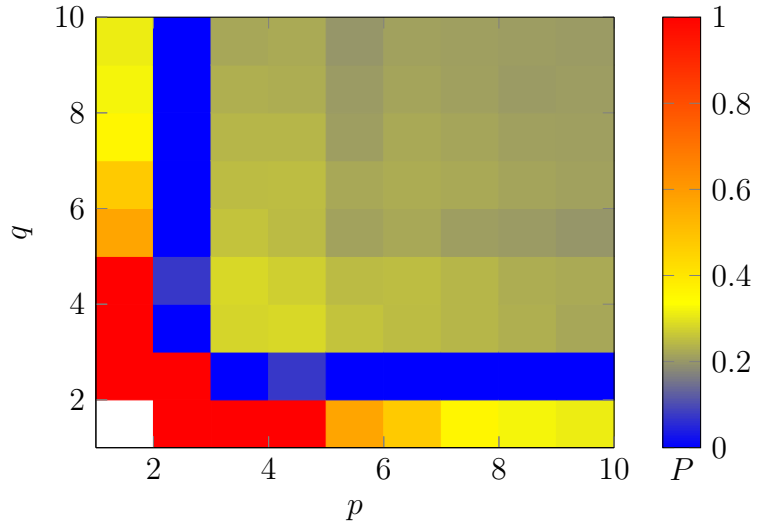


Figure 9.1

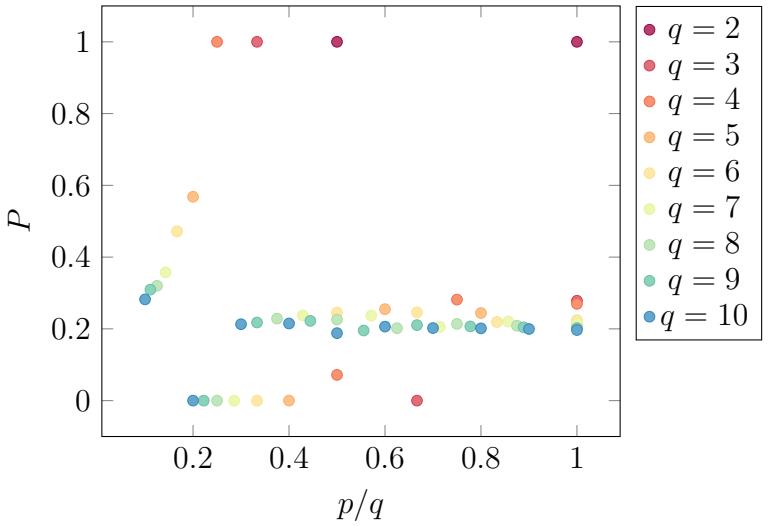


Figure 9.2

We now analyse our results. Throughout our analysis, we take $p \leq q$ for brevity. We split our analysis into three sections, each focusing on a different trend in our data. In particular, we will be discussing the cases when $p = 1$, $p = 2$ and $p > 2$, which can be clearly distinguished on Figure 9.2.

9.1 $p = 1$ Trend

As one can observe from Figure 9.1, P is inversely related to q when $p = 1$. To investigate this trend, we calculate the critical densities of all $1 \times q$ Minesweeper

board with $q \leq 50$ and plot them on Figure 9.3. We then fit the resulting curve with the series $\sum_{n=0}^{\infty} a_n q^{-n}$.

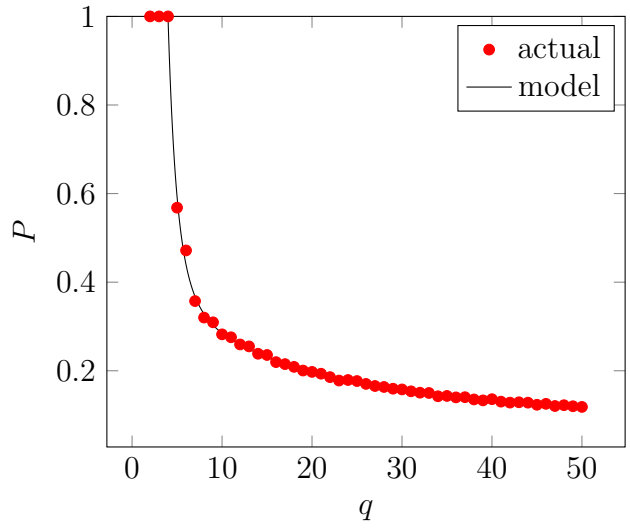


Figure 9.3

As $q \to \infty$, we have $P \to a_0$. Plotting a_0 for the first k partial sums yields Figure 9.4.

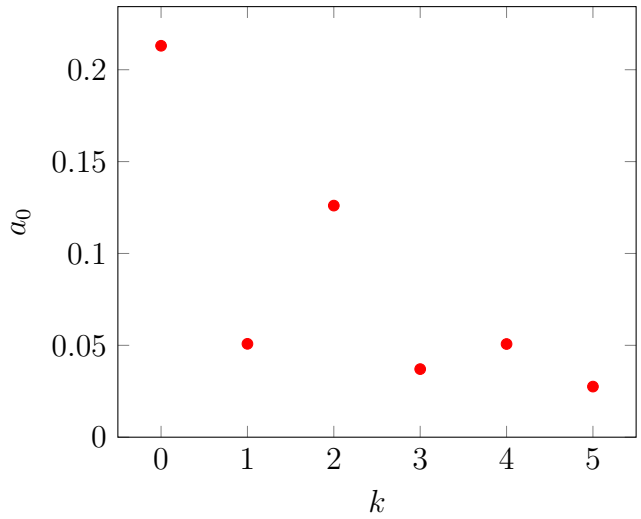


Figure 9.4

We thus conjecture that $a_0 \to 0$ as $k \to \infty$, whence $P \to 0$ as $q \to \infty$ for $p = 1$.

9.2 $p = 2$ Trend

When $p = 2$, one can observe that $P = 0$ for almost all q. A closer look at the win rate data of these board sizes reveals the reason behind this behaviour.

Figure 9.5 plots the win rate of a 2×10 board. We observe that due to the narrowness of the board, all clicks are practically 50/50 guesses. Thus, the win rate starts off at around 0.50 and immediately begins to decrease. This makes the win rate trend appear to be a sigmoid curve that has been shifted significantly to the left. Hence, Q is either less than or close to 0. This results in $P = 0$ for most q.

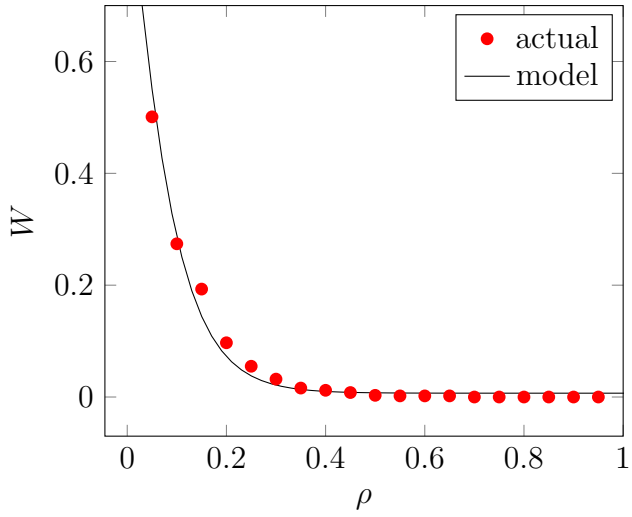


Figure 9.5

9.3 $p > 2$ Trend

From Figure 9.2, one can observe that for $p \times q$ boards where $p > 2$, the critical density P remains stable at approximately 0.20 regardless of the aspect ratio $\frac{p}{q}$. We also note that P decreases as q increases. This inverse relationship is especially obvious when $\frac{p}{q} = 1$. This suggests that there exists some asymptotic critical density $P^\star = \lim_{q\to\infty} P$ that is the same for all aspect ratios $\frac{p}{q}$.

We now approximate $P^\star$. We begin by plotting P for all $q \times q$ boards and model the resulting plot with the function $\frac{a}{q-b} + c$ for some constants a, b and c. This is illustrated in Figure 9.6.

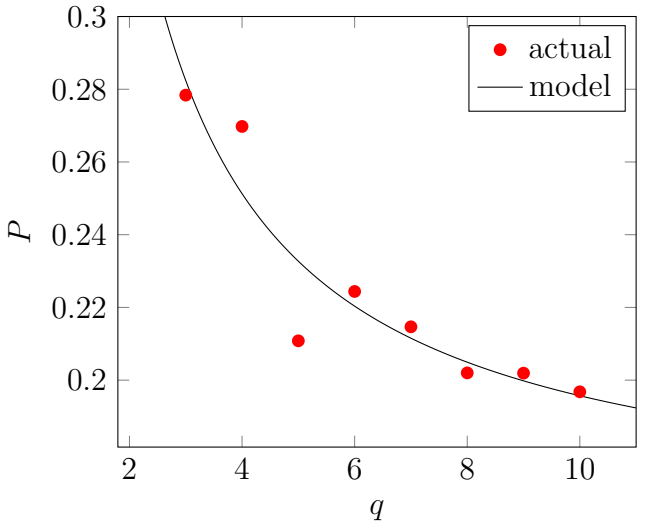


Figure 9.6

Our model returns $a = 0.366581$, $b = 0.0305678$ and $c = 0.158938$ as the parameters for the best-fit plot. We thus have that $P^\star$ is approximately equal to 0.159.

10 Conclusion

In sum, we have developed an original Minesweeper solver that can infer logically and guess smartly. We also devised an original algorithm to calculate the critical density of any board size. We then used the algorithm to calculate the critical density of all board sizes smaller than 10×10. Finally, we analysed our results and concluded that the critical density converges to approximately 0.159 for all board sizes where $p > 2$. We also demonstrated that the critical density converges to 0 for all board sizes where $p \leq 2$.

10.1 Applications

Our results provide a metric for the difficulty of Minesweeper boards. For sufficiently large boards, below the critical density, players can expect the board to be trivially easy to solve, while above the critical density, players can expect the board to be near-impossible to solve. However, near the critical density, the board would be challenging for the player, but not outright impossible. This sweet-spot would allow players to improve on their Minesweeper abilities, such as their inference and guessing skills. Hence, our results provide a valuable resource for players to maximise their training efficiency, all while having fun tackling a board that is neither too easy nor too hard.

10.2 Limitations

The biggest limitation of our research is our Minesweeper solver; when compared to more popular solvers, our solver is not as good in terms of win rate and efficiency. For example, Hill's solver [8] is able to solve 10 thousand 10×10 / 20 boards with a win rate of 61.1% in approximately 20 seconds. In comparison, our solver can only solve a thousand boards with a win rate of 46.8% in 4 minutes.

By using a stronger solver, the win rates of almost all boards would improve, leading to an increase in the critical densities of almost all dimensions. Furthermore, the critical densities of larger board sizes could be calculated. Overall, this would result in a more accurate calculation of the asymptotic critical density $P^\star$. This explains why our calculated value for $P^\star$ is significantly lower than the 0.20 observed by Demsey and Guinn [1] and Sinha et al. [2] We believe that a stronger solver would give an asymptotic critical density that more closely matches the aforementioned observations.

10.3 Future Directions

We have identified two possible ways to expand on our research.

Firstly, the critical density of Minesweeper variants can be analysed by adapting our methodology. Such variants include higher dimensional boards and boards with non-square tiles. We posit that the asymptotic critical density $P^\star$ could serve as a measure for the complexity of Minesweeper variants; the lower the critical density, the more complex the variant is.

Secondly, one could also consider different definitions for the critical density. In this paper, we defined the critical density in terms of win rate. However, the critical

density could be based on other metrics, such as the time taken to solve a board, or average number of guesses taken.

11 References

[1] R. Dempsey and C. Guinn. A Phase Transition in Minesweeper, 2020. arXiv:2008.04116 [cs.AI].

[2] Y. P. Sinha, P. Malviya, and R. K. Nayak. Fast constraint satisfaction problem and learning-based algorithm for solving Minesweeper, 2021. arXiV:2105.04120 [cs.AI].

[3] Y. Q. Qing, W. L. You, and M. X. Liu. Critical exponents and the universality class of a minesweeper percolation model, 2020. doi:10.1142/S0129183120501296.

[4] P. Crow. A Mathematical Introduction to the Game of Minesweeper. *Undergraduate Mathematics and Its Applications Journal*, 18(1), 1997.

[5] K. Huang. A Human's Guide to Minesweeper, 2023.

[6] D. Becerra. Algorithmic Approaches to Playing Minesweeper, 2015.

[7] J. E. Dennis and H. H. W. Mei. An Unconstrained Optimization Algorithm which uses Function and Gradient Values, 1975.

[8] D. N. Hill. Minesweeper2. https://github.com/DavidNHill/Minesweeper2, 2019.

[9] M. de Bondt. The computational complexity of Minesweeper, 2012. arXiv:1204.4659 [cs.CC].

[10] R. Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22(2):9–15, 2000. doi:10.1007/BF03025367.

A Code

A.1 Solver

The following files have been reproduced here on the basis that they are important components to the solver. The complete source code can be accessed via `github.com/asdia0/Minesweeper`.

Inferrer.cs

```
using System.Collections.Generic;
using System.Linq;

namespace Minesweeper.Solver
{
    public class Inferrer
    {
        public HashSet<Constraint> Constraints { get; set; }

        public HashSet<Constraint> Solutions { get; set; }

        public bool HasContradiction { get; set; }

        public Inferrer(Grid grid)
        {
            this.Constraints = [];
            this.Solutions = [];

            foreach (Cell boundaryCell in grid.BoundaryCells)
            {
                HashSet<int> cellVariables = Utility.CellsToIDs(
                    boundaryCell.AdjacentCells.Intersect(grid.
                    UnknownCells)).ToHashSet();
                this.Constraints.Add(new(cellVariables, (int)
                    boundaryCell.RemainingMineCount));
            }
        }

        public void Solve()
        {
            bool run = true;

            List<Constraint> oldConstraints = [];

            while (run)
            {
                this.SolveTrivials();
                this.ConstructConstraints();
                this.RemoveUnnecessaryConstraints();
                this.UpdateSolutions();

                bool runTemp = this.Constraints.Except(
                    oldConstraints).Any() && !this.HasContradiction;
                oldConstraints = [.. this.Constraints];
                run = runTemp;
            }
        }

```

```
45          public void SolveTrivials()
46          {
47              HashSet<Constraint> trivialAllSafe = this.Constraints.
                    Where(i => i.Sum == 0).ToHashSet();
48              HashSet<Constraint> trivialAllMined = this.Constraints.
                    Where(i => i.Sum == i.Variables.Count).ToHashSet();
49
50              foreach (Constraint constraint in trivialAllSafe)
51              {
52                  foreach (int variable in constraint.Variables)
53                  {
54                      this.Solutions.Add(new([variable], 0));
55                  }
56
57                  this.Constraints.Remove(constraint);
58              }
59
60              foreach (Constraint constraint in trivialAllMined)
61              {
62                  foreach (int variable in constraint.Variables)
63                  {
64                      this.Solutions.Add(new([variable], 1));
65                  }
66
67                  this.Constraints.Remove(constraint);
68              }
69          }
70
71          public void RemoveUnnecessaryConstraints()
72          {
73              this.Constraints.RemoveWhere(i => i.Variables.Count ==
                    0);
74          }
75
76          public void ConstructConstraints()
77          {
78              int constraintCount = this.Constraints.Count;
79
80              List<Constraint> constraintList = [.. this.Constraints
                    ];
81
82              for (int i = 0; i < constraintCount; i++)
83              {
84                  for (int j = 0; j < constraintCount; j++)
85                  {
86                      Constraint X = constraintList[i];
87                      Constraint Y = constraintList[j];
88
89                      bool canSubtract = X.Subtract(Y, out Constraint
                            difference);
90
91                      if (canSubtract)
92                      {
93                          this.HasContradiction = difference.Sum < 0;
94                          this.Constraints.Add(difference);
95                      }
96                  }
97              }
```

```
98          }
99
100         public void UpdateSolutions()
101         {
102             this.Solutions.UnionWith(this.Constraints.Where(i => i.
                    IsSolved));
103             this.Constraints = this.Constraints.Except(this.
                    Solutions).ToHashSet();
104
105             foreach (Constraint solution in this.Solutions.Distinct
                    ())
106             {
107                 int ID = solution.Variables.First();
108
109                 foreach (Constraint constraint in this.Constraints)
110                 {
111                     bool solutionPresent = constraint.Variables.
                            Remove(ID);
112
113                     if (solutionPresent)
114                     {
115                         constraint.Sum -= solution.Sum;
116                     }
117                 }
118             }
119         }
120     }
121 }
```

Guesser.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4
5  namespace Minesweeper.Solver
6  {
7      public class Guesser
8      {
9          public Grid Grid { get; set; }
10
11         public HashSet<Constraint> Constraints { get; set; }
12
13         public Guesser(Grid grid)
14         {
15             this.Grid = grid;
16             this.Constraints = [];
17
18             foreach (Cell boundaryCell in grid.BoundaryCells)
19             {
20                 HashSet<int> cellVariables = [.. Utility.CellsToIDs
                       (boundaryCell.AdjacentCells.Intersect(grid.
                       UnknownCells))];
21                 Constraints.Add(new(cellVariables, (int)
                       boundaryCell.RemainingMineCount));
22             }
23         }
24
25         public HashSet<HashSet<Constraint>> GetGroups(HashSet<
               Constraint> constraints)
26         {
27             HashSet<HashSet<Constraint>> results = [];
28
29             HashSet<HashSet<Constraint>> preliminaryGroups =
                   GetPreliminaryGroups(constraints);
30
31             foreach (HashSet<Constraint> preliminaryGroup in
                   preliminaryGroups)
32             {
33                 results.UnionWith(GetFinalGroups(preliminaryGroup))
                       ;
34             }
35
36             results.RemoveWhere(i => i.Count == 0);
37
38             return results;
39         }
40
41         public HashSet<HashSet<Constraint>> GetPreliminaryGroups(
               HashSet<Constraint> constraints)
42         {
43             HashSet<HashSet<Constraint>> groups = [];
44
45             HashSet<Constraint> remainingConstraints = [..
                   constraints];
46
47             HashSet<Constraint> toSearch = [];
48             HashSet<Constraint> group = [];
```

```
49              HashSet<Constraint> searched = [];
50
51              while (remainingConstraints.Count > 0)
52              {
53                  Constraint seed = null;
54
55                  if (toSearch.Count != 0)
56                  {
57                      seed = toSearch.First();
58                  }
59                  else
60                  {
61                      groups.Add(group);
62                      group = [];
63                      seed = remainingConstraints.First();
64                  }
65
66                  foreach (int id in seed.Variables)
67                  {
68                      toSearch.UnionWith(remainingConstraints.Where(i
                            => i.Variables.Contains(id)));
69                  }
70
71                  toSearch = toSearch.Except(searched).ToHashSet();
72
73                  group.Add(seed);
74
75                  searched.Add(seed);
76                  remainingConstraints.Remove(seed);
77              }
78
79              groups.Add(group);
80
81              return groups;
82          }
83
84          public HashSet<HashSet<Constraint>> GetFinalGroups(HashSet<
                Constraint> constraints)
85          {
86              HashSet<Constraint> constraintsTemp = [.. constraints];
87              HashSet<HashSet<Constraint>> groups = [];
88
89              HashSet<HashSet<int>> intersections = Utility.GetGroups
                    (constraints.Select(i => i.Variables).ToHashSet());
90
91              foreach (HashSet<int> intersection in intersections)
92              {
93                  Constraint constraint = constraintsTemp.Where(i =>
                        i.Variables == intersection).FirstOrDefault();
94
95                  if (constraint is null)
96                  {
97                      continue;
98                  }
99
100                 HashSet<Constraint> constraintSupersets =
                        constraintsTemp.Where(i => i.Variables.
                        IsProperSupersetOf(intersection)).ToHashSet();
```

```
101
102                 foreach (Constraint constraintSuperset in
                        constraintSupersets)
103                 {
104                     constraintsTemp.Remove(constraintSuperset);
105                     constraintsTemp.Add(new(constraintSuperset.
                            Variables.Except(intersection).ToHashSet(),
                            constraintSuperset.Sum - constraint.Sum));
106                 }
107
108                 groups.Add([constraint]);
109             }
110
111             groups.Add(constraintsTemp);
112
113             return groups;
114         }
115
116         public HashSet<Configuration> GetGroupConfigurations(
                Configuration seed, int depth = 0, int maxDepth = 5)
117         {
118             HashSet<Configuration> configs = [];
119
120             List<int> variables = [.. seed.Assignments.Keys];
121             List<int> unsolvedVariables = [.. seed.Assignments.
                    Where(i => i.Value == null).Select(i => i.Key)];
122             List<int> solvedVariables = [.. seed.Assignments.Where(
                    i => i.Value != null).Select(i => i.Key)];
123
124             if (unsolvedVariables.Count == 0)
125             {
126                 return [seed];
127             }
128
129             int ID = unsolvedVariables.First();
130
131             Inferrer solverSafe = new(this.Grid);
132             foreach (int solvedVariable in solvedVariables)
133             {
134                 solverSafe.Constraints.Add(new Constraint([
                        solvedVariable], (int)seed.Assignments[
                        solvedVariable]));
135             }
136             solverSafe.Constraints.Add(new Constraint([ID], 0));
137
138             solverSafe.Solve();
139
140             Configuration newConfigurationSafe = new(variables,
                    solverSafe.Solutions
141                         .Where(i => variables.Contains(i.Variables.
                                First()))
142                         .ToHashSet());
143
144             if (!newConfigurationSafe.Assignments.Where(i => i.
                    Value < 0).Any())
145             {
146                 if (newConfigurationSafe.IsSolved)
147                 {
```

```
148                     configs.Add(newConfigurationSafe);
149                 }
150                 else if (depth <= maxDepth)
151                 {
152                     configs.UnionWith(GetGroupConfigurations(
                           newConfigurationSafe, depth + 1));
153                 }
154                 else
155                 {
156                     configs.UnionWith([GetOneConfiguration(
                           newConfigurationSafe)]);
157                 }
158             }
159 
160             Inferrer solverMined = new(this.Grid);
161             foreach (int solvedVariable in solvedVariables)
162             {
163                 solverMined.Constraints.Add(new Constraint([
                       solvedVariable], (int)seed.Assignments[
                       solvedVariable]));
164             }
165             solverMined.Constraints.Add(new Constraint([ID], 1));
166 
167             solverMined.Solve();
168 
169             Configuration newConfigurationMined = new(variables,
                   solverMined.Solutions
170                     .Where(i => variables.Contains(i.Variables.
                           First()))
171                     .ToHashSet());
172 
173             if (!newConfigurationMined.Assignments.Where(i => i.
                   Value < 0).Any())
174             {
175                 if (newConfigurationMined.IsSolved)
176                 {
177                     configs.Add(newConfigurationMined);
178                 }
179                 else if (depth <= maxDepth)
180                 {
181                     configs.UnionWith(GetGroupConfigurations(
                           newConfigurationMined, depth + 1));
182                 }
183                 else
184                 {
185                     configs.UnionWith([GetOneConfiguration(
                           newConfigurationMined)]);
186                 }
187             }
188 
189             return configs.Where(i => i.Assignments != null).
                   ToHashSet();
190         }
191 
192         public Configuration GetOneConfiguration(Configuration seed
               )
193         {
194             List<int> variables = [.. seed.Assignments.Keys];
```

```
195             List<int> unsolvedVariables = [.. seed.Assignments.
                    Where(i => i.Value == null).Select(i => i.Key)];
196             List<int> solvedVariables = [.. seed.Assignments.Where(
                    i => i.Value != null).Select(i => i.Key)];
197
198             int ID = unsolvedVariables.First();
199
200             Inferrer solverSafe = new(this.Grid);
201
202             foreach (int solvedVariable in solvedVariables)
203             {
204                 solverSafe.Constraints.Add(new Constraint([
                        solvedVariable], (int)seed.Assignments[
                        solvedVariable]));
205             }
206
207             solverSafe.Constraints.Add(new Constraint([ID], 0));
208
209             solverSafe.Solve();
210
211             if (!solverSafe.HasContradiction)
212             {
213                 Configuration newConfigurationSafe = new(variables,
                        solverSafe.Solutions
214                         .Where(i => variables.Contains(i.Variables.
                            First()))
215                         .ToHashSet());
216
217                 if (newConfigurationSafe.IsSolved)
218                 {
219                     return newConfigurationSafe;
220                 }
221                 else
222                 {
223                     return GetOneConfiguration(newConfigurationSafe
                            );
224                 }
225             }
226
227             Inferrer solverMined = new(this.Grid);
228
229             foreach (int solvedVariable in solvedVariables)
230             {
231                 solverMined.Constraints.Add(new Constraint([
                        solvedVariable], (int)seed.Assignments[
                        solvedVariable]));
232             }
233
234             solverMined.Constraints.Add(new Constraint([ID], 1));
235
236             solverMined.Solve();
237
238             if (!solverMined.HasContradiction)
239             {
240                 Configuration newConfigurationMined = new(variables
                        , solverMined.Solutions
241                         .Where(i => variables.Contains(i.Variables.
                            First()))
```

```
242                     .ToHashSet());
243
244                 if (newConfigurationMined.IsSolved)
245                 {
246                     return newConfigurationMined;
247                 }
248                 else
249                 {
250                     return GetOneConfiguration(
                            newConfigurationMined);
251                 }
252             }
253
254             return new();
255         }
256
257         public Dictionary<int, double> GetSafety()
258         {
259             if (Grid.ExposedCells.Count == 0)
260             {
261                 return this.Grid.UnknownCells.ToDictionary(key =>
                        key.Point.ID, value => 1 - (double)(this.Grid.
                        RemainingMines) / this.Grid.UnknownCells.Count);
262             }
263
264             List<HashSet<Configuration>> groupConfigurations = [];
265
266             foreach (HashSet<Constraint> group in this.GetGroups(
                    this.Constraints))
267             {
268                 Configuration config = new(group.SelectMany(i => i.
                        Variables).Distinct().ToList(), []);
269                 HashSet<Configuration> groupConfiguration = this.
                        GetGroupConfigurations(config);
270                 groupConfiguration.RemoveWhere(i => i.Assignments.
                        Values.Where(i => i < 0).Any());
271                 groupConfigurations.Add([.. groupConfiguration]);
272             }
273
274             Dictionary<int, double> safetyValues = this.Grid.
                    UnknownCells.ToDictionary(key => key.Point.ID, value
                     => (double)0);
275
276             Dictionary<Configuration, double> weights = [];
277
278             foreach (HashSet<Configuration> groupConfiguration in
                    groupConfigurations)
279             {
280                 foreach (Configuration configuration in
                        groupConfiguration)
281                 {
282                     weights.Add(configuration, Utility.Choose(this.
                            Grid.FloatingCells.Count, this.Grid.
                            RemainingMines - configuration.Sum));
283                 }
284             }
285
286             double denominator = weights.Values.Sum();
```

```
287
288             foreach (HashSet<Configuration> groupConfiguration in
                    groupConfigurations)
289             {
290                 foreach (Configuration configuration in
                        groupConfiguration)
291                 {
292                     foreach (int exposedCell in Utility.CellsToIDs(
                            this.Grid.ExposedCells).Intersect(
                            configuration.Assignments.Where(i => i.Value
                             == 0).Select(i => i.Key)))
293                     {
294                         safetyValues[exposedCell] += weights[
                                configuration];
295                     }
296                 }
297             }
298
299             foreach (int exposedCell in safetyValues.Keys)
300             {
301                 safetyValues[exposedCell] = safetyValues[
                        exposedCell] / denominator;
302             }
303
304             double expectedFloatingMines = this.Grid.RemainingMines
                     - safetyValues.Count + safetyValues.Values.Sum();
305
306             double floatingSafety = 1 - expectedFloatingMines /
                    this.Grid.FloatingCells.Count;
307
308             foreach (int floatingCell in Utility.CellsToIDs(this.
                    Grid.FloatingCells))
309             {
310                 safetyValues[floatingCell] = floatingSafety;
311             }
312
313             return safetyValues;
314         }
315     }
316 }
```

Solver.cs

```
1  public static int Solve(Grid grid)
2  {
3      grid.OpenCell(grid.Cells[0]);
4
5      while (grid.State == State.Ongoing)
6      {
7          Inferrer solver = new(grid);
8          solver.Solve();
9
10         if (solver.Solutions.Count != 0)
11         {
12             foreach (Constraint solution in solver.Solutions)
13             {
14                 Cell cell = Utility.IDToCell(grid, solution.
                       Variables.First());
15
16                 switch (solution.Sum)
17                 {
18                     case 0:
19                         grid.OpenCell(cell);
20                         break;
21                     case 1:
22                         grid.FlagCell(cell);
23                         break;
24                     default:
25                         throw new MinesweeperException("Invalid
                               solution: " + solution);
26                 }
27             }
28         }
29         else
30         {
31             Guesser guesser = new(grid);
32             Dictionary<int, double> scores = guesser.GetSafety();
33
34             foreach (Cell safeCells in Utility.IDsToCells(grid,
                   scores.Where(i => i.Value == 1).Select(i => i.Key)))
35             {
36                 grid.OpenCell(safeCells);
37             }
38
39             double maxScore = scores.OrderByDescending(kvp => kvp.
                   Value).First().Value;
40
41             Cell toOpen = Utility.IDsToCells(grid, scores.Where(i
                   => i.Value == maxScore).Select(i => i.Key))
42                 .OrderBy(i => i.AdjacentCells.Count)
43                 .ThenByDescending(i => i.AdjacentCells.Intersect(
                       grid.OpenedCells).Count())
44                 .First();
45
46             grid.OpenCell(toOpen);
47         }
48     }
49
50     if (grid.State == State.Success)
51     {
```

```
52          return 1;
53      }
54      else
55      {
56          return 0;
57      }
58  }
```

A.2 Critical Density

CriticalDensity.py

```
from scipy.optimize import curve_fit
import numpy as np
import os
import matplotlib.pyplot as plt

class Dimension:
    def __init__(self, p, q):
        self.p = min(p, q)
        self.q = max(p, q)

    def __str__(self):
        return str(self.p) + str(self.q)

    def __eq__(self, other):
        if not isinstance(other, Dimension):
            raise NotImplementedError

        return self.p == other.p and self.q == other.q

    def __hash__(self):
        return hash(str(self))

directory = os.getcwd() + "/docs"
winRates = {}
criticalDensities = {}

def LogisticFunction(x, b, L, k, Q):
    return b + L / (1 + np.exp(k * (x-Q)))

def GetLogisticParameters(dimension, xdata, ydata):
    xdata = np.array(xdata)
    ydata = np.array(ydata)

    initialGuesses = [min(ydata), max(ydata) - min(ydata), 10, np.
        median(xdata)]
    param_bounds=([-np.inf,0,0,-np.inf],[np.inf,3,np.inf,np.inf])

    parameters, covariance = curve_fit(LogisticFunction, xdata,
        ydata, p0=initialGuesses, bounds=param_bounds, method='
        dogbox', maxfev=1000000)

    return [*parameters]

def LoadWinRates(maxDimension):
    for num in range(2, maxDimension + 1):
        with open(f'{directory}/WinRates/{num}.csv', "r") as f:
            for line in f.readlines():
                data = line.split(",")

                p = int(data[0])
                q = int(data[1])
                m = int(data[2])
                winRate = float(data[3])

```

```
52                  mineDensity = m/(p*q)
53                  dimension = Dimension(p, q)
54
55                  if mineDensity > 0.75 and winRates[dimension
                        ][1][-1] < winRate:
56                      winRate = 0
57
58                  if dimension in winRates:
59                      winRates[dimension][0].append(mineDensity)
60                      winRates[dimension][1].append(winRate)
61
62                  else:
63                      winRates[dimension] = [[mineDensity], [winRate
                            ]]
64
65  def CalculateCriticalDensities():
66   for dimension in winRates:
67      mineDensity = winRates[dimension][0]
68      winRate = winRates[dimension][1]
69
70      mineDensity.insert(0, 0)
71      winRate.insert(0, 1)
72
73      mineDensity.append(1)
74      winRate.append(0)
75
76      parameters = GetLogisticParameters(dimension, mineDensity,
            winRate)
77      criticalDensity = parameters[3]
78
79      if criticalDensity < 0:
80          criticalDensity = 0
81      elif criticalDensity > 1:
82          criticalDensity = 1
83
84      criticalDensities[dimension] = criticalDensity
85
86  def WriteCriticalDensities(maxDimension):
87      with open(f'{directory}/CriticalDensities.csv', "w") as g:
88          g.write("p,q,P\n")
89
90          for p in range(1, maxDimension + 1):
91              for q in range(1, maxDimension + 1):
92                  dimension = Dimension(p, q)
93
94                  if dimension == Dimension(1, 1):
95                      g.write("1,1,nan\n")
96                      continue
97
98                  g.write(f"{p},{q},{criticalDensities[dimension]}\n"
                        )
99                  g.flush()
100
101 def RebaseToAspectRatio(maxDimension):
102     for i in range(2, maxDimension + 1):
103         ratioDensityDict = {}
104
105         for dimension in [j for j in criticalDensities if i == j.q
```

```
            ]:
106             ratioDensityDict[dimension.p/i] = criticalDensities[
                    dimension]
107
108         with open(f'{directory}/RatioCD/{i}.csv', "w") as g:
109             g.write("ratio,CD\n")
110
111             for aspectRatio in ratioDensityDict:
112                 g.write(f"{aspectRatio},{ratioDensityDict[
                        aspectRatio]}\n")
113                 g.flush()
114
115 if __name__ == "__main__":
116     maxDimension = 10
117
118     LoadWinRates(maxDimension)
119     CalculateCriticalDensities()
120     WriteCriticalDensities(maxDimension)
121     RebaseToAspectRatio(maxDimension)
```

B Data

B.1 Win Rate

Due to the size of the win rate data, it is not printed in this paper and can only be accessed via `asdia.dev/minesweeper-critical-density/win-rates.csv`.

B.2 Critical Density

p	q	P	p	q	P
1	2	1	4	5	0.24417529634429253
1	3	1	4	6	0.24611222387092954
1	4	1	4	7	0.23715170452873602
1	5	0.5681473499112628	4	8	0.2258446843926504
1	6	0.4716362057677376	4	9	0.22227246964036515
1	7	0.3573598239211957	4	10	0.21501775629655295
1	8	0.3200466082058473	5	5	0.21081575907312486
1	9	0.30940463606882607	5	6	0.21912865353261837
1	10	0.2822166976806722	5	7	0.20552189546717414
2	2	1	5	8	0.20190924155988516
2	3	0	5	9	0.19542866372751225
2	4	0.07172544132146422	5	10	0.18807656414318988
2	5	0	6	6	0.22438667018416458
2	6	0	6	7	0.22028216887894334
2	7	0	6	8	0.21389993159016554
2	8	0	6	9	0.21044808514885935
2	9	0	6	10	0.20625288737080402
2	10	0	7	7	0.21466887936812817
3	3	0.27837876338731626	7	8	0.20858547016901607
3	4	0.28151285279312666	7	9	0.20689137479962186
3	5	0.2548740484439963	7	10	0.2022594055980943
3	6	0.24522567024938602	8	8	0.20199382622106984
3	7	0.23751576681494638	8	9	0.20434960152369053
3	8	0.22864034399741218	8	10	0.20140219935313744
3	9	0.2178173885321011	9	9	0.2019203353155787
3	10	0.21296825578136416	9	10	0.19966867277772682
4	4	0.26976512506230443	10	10	0.19679192187271458