# SAT Modeling and SAT Solver Implementation for Nonograms

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Group
https://ai.dmi.unibas.ch/

Examiner: Prof. Malte Helmert
Supervisor: Claudia Grundke

Aldris Arslani
aldris.arslani@unibas.ch
21-051-842

August 20, 2025

**Abstract**

This thesis investigates automated Nonogram solving through propositional satisfiability (SAT). Nonograms are logic puzzles defined by numerical row and column clues that specify consecutive blocks of black cells in a grid. We present two encodings of Nonograms as conjunctive normal form (CNF) formulas: the Sequence Enumeration encoding, which models each valid coloring of a line as a variable, and the Block-Based encoding, which models each block of a valid coloring as a variable. To evaluate these encodings, we implemented a conflict-driven clause learning (CDCL) solver with specialized decision heuristics for Nonograms. Comparative experiments demonstrate that the Block-Based encoding yields smaller formulas and faster solving times than Sequence Enumeration. Furthermore, heuristics exploiting structural properties of Nonograms can outperform general-purpose heuristics. A comparison with MiniSat indicates that overall performance remains largely determined by the efficiency of the implementation.

# Contents

# 1 Introduction

Nonograms are logic puzzles in which a hidden black-and-white image is reconstructed on a grid using numerical clues. Each clue specifies the lengths of consecutive black cells in a row or column, while all other cells remain white. By combining the information from all clues, the complete image can be determined. An example puzzle together with its solution is shown in Figure 1.

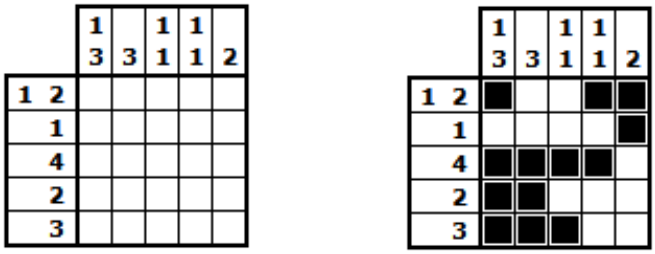



Figure 1: A Nonogram puzzle (left) and its unique solution (right).

Nonograms can be described formally and encoded as instances of the Satisfiability Problem (SAT). Different encodings have been explored, ranging from straightforward formulations that directly model each cell to approaches based on deterministic finite automata that capture whole line descriptions [7]. In this thesis we present two SAT encodings of Nonograms that are both based on recent Constraint Satisfaction Problem (CSP) approaches introduced by Aramian and Yeghiazaryan [2]. Our encodings retain the structural ideas of those CSP approaches but translate them into Conjunctive Normal Form (CNF) formulas suitable for SAT solvers.

To solve these formulas we require a SAT solver. For this purpose we implemented our own solver from scratch. The implementation follows the basic design of modern conflict-driven clause learning (CDCL) solvers, taking the SAT solver MiniSat [6] as a minimal reference point and relying on the description of CDCL in [11]. This solver provides us with full control over the internals, which allows us to implement specialized decision heuristics for our Nonogram encodings. This gives us the opportunity to not only evaluate the encodings themselves, but also to study whether heuristics that exploit the structure of Nonograms can accelerate the solving process.

# 2 Background

In order to present the methods and implementation used in this thesis, we first give an introduction to propositional logic, focusing on CNF formulas and their evaluation under partial assignments. We then describe the central techniques of modern SAT solvers, in particular unit propagation and CDCL. Lastly, we present the DIMACS format, the standard representation of CNF formulas used as input for SAT solvers.

## 2.1 CNF Formulas

We introduce the basic notation and terminology of propositional logic used in this thesis, based on [11, 9]. The presentation is tailored to the purpose of CNF encodings and SAT solvers, and should be understood in that context.

We always assume a finite set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$. We use the terms *variable* and *Boolean variable* interchangeably in this thesis.

A literal is either a variable $x_i \in X$ or its negation $\neg x_i$. A clause is a disjunction of one or more literals, for example $(x_1 \vee \neg x_3 \vee x_4)$. A formula is a conjunction of such clauses, for example $(x_1) \wedge (x_2 \vee \neg x_3)$. We also use the terms *formula* and *CNF formula* interchangeably.

A formula can equivalently be written in set notation. Each clause is viewed as a finite set of literals, and the formula as a finite set of such clauses. For example, the formula $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4)$ can be written as the set $\{\{x_1, \neg x_2\},\ \{\neg x_1, x_3, x_4\}\}$. The notation used will be clear from context.

An assignment is a function $\tau : X \to \{0, u, 1\}$, where $u$ denotes an undefined value. If all variables are assigned either 0 or 1 under $\tau$, the assignment is called a total assignment. If at least one variable is assigned $u$, the assignment is partial.

In this thesis, extending an assignment refers to creating a new assignment $\tau'$ from an existing assignment $\tau$ by setting $\tau'(x_i) := v$ for some variable $x_i$ with $\tau(x_i) = u$ and $v \in \{0, 1\}$, such that $\tau'(x_j) = \tau(x_j)$ for all $x_j \neq x_i$. For simplicity, we refer to the extended assignment again as $\tau$.

Assignments are used to evaluate literals, clauses, and formulas. We define $0 < u < 1$ and $1 - u = u$. Then, given an assignment $\tau$, the value of a literal $l$ under $\tau$ is defined as

$$l^\tau = \begin{cases} \tau(x_i) & \text{if } l = x_i \\ 1 - \tau(x_i) & \text{if } l = \neg x_i. \end{cases}$$

The value of a clause $\omega$ under $\tau$ is defined as the maximum of the values of its literals:

$$\omega^\tau = \max\{l^\tau \mid l \in \omega\}.$$

The value of a formula $\varphi$ under $\tau$ is defined as the minimum of the values of its clauses:

$$\varphi^\tau = \min\{\omega^\tau \mid \omega \in \varphi\}.$$

We call a clause $\omega$ satisfied by an assignment $\tau$ if $l^\tau = 1$ for some $l \in \omega$, and falsified if $l^\tau = 0$ for all $l \in \omega$. A formula $\varphi$ is satisfiable if there exists a assignment $\tau$ such that $\omega^\tau = 1$ for all $\omega \in \varphi$, and unsatisfiable otherwise. We note that an assignment $\tau$ for which $\varphi^\tau = 1$ can be partial, since every clause $\omega$ requires only one of its literals to evaluate to 1 for $\omega^\tau = 1$ to hold.

## 2.2 SAT Solvers

The propositional satisfiability problem is the task of deciding whether a CNF formula $\varphi$ is satisfiable. A tool that solves this problem is called a SAT solver. It takes $\varphi$ as input and either reports that $\varphi$ is satisfiable and returns an assignment $\tau$ such that $\varphi^\tau = 1$, or reports that $\varphi$ is unsatisfiable.

One approach to solving the SAT problem is based on the CDCL algorithm. While representations of its components may vary, we use a representation based of [11, 9].

The CDCL algorithm uses a technique known as unit propagation. Given a CNF formula $\varphi$ and an assignment $\tau$, a clause $\omega = (l_1 \vee \cdots \vee l_k)$ in $\varphi$ is said to be unit under $\tau$ if $l_j^\tau = u$ for exactly one literal $l_j$ with $1 \leq j \leq k$, and $l_i^\tau = 0$ for all $i = 1, \ldots, k$ with $i \neq j$. In this case, $l_j$ is said to be implied by $\omega$ under $\tau$, and $\tau$ is extended by setting $\tau(x_j) := 1$ if $l_j = x_j$, or $\tau(x_j) := 0$ if $l_j = \neg x_j$, assuming $l_j \in \{x_j, \neg x_j\}$. This step is referred to as a unit propagation step.

A single unit propagation step can cause other clauses in the formula to become unit, allowing further unit propagation steps, or cause a clause to become falsified. In the latter case, we say that a conflict has occurred. The process of repeatedly applying unit propagation steps until either a clause is falsified or no unit clause remains is called unit propagation. We note that if no clause is unit under the assignment, then no unit propagation step can be performed.

**Example.** Let $\varphi = \{\{x_1, x_2\}, \{\neg x_2, x_3\}, \{\neg x_3\}\}$ and let $\tau$ be an assignment such that $\tau(x_1) = 0$, $\tau(x_2) = u$, and $\tau(x_3) = u$. We now perform unit propagation, considering the clauses from left to right.

The first clause $\{x_1, x_2\}$ is unit under $\tau$ because $x_1^\tau = 0$ and $x_2^\tau = u$. Thus, $x_2$ is implied and $\tau$ is extended with $\tau(x_2) := 1$, making $x_2^\tau = 1$.

The second clause $\{\neg x_2, x_3\}$ is now unit because $(\neg x_2)^\tau = 0$ and $x_3^\tau = u$. Therefore, $x_3$ is implied and $\tau$ is extended with $\tau(x_3) := 1$, giving $x_3^\tau = 1$.

The third clause $\{\neg x_3\}$ is now falsified since $(\neg x_3)^\tau = 0$. A conflict therefore occurs.

A decision is an extension of an assignment made without a clause implying it, where a variable $x$ whose assigned value is undefined is assigned a value such that the literal $l \in \{x, \neg x\}$ evaluates to 1.

A trail is a sequence of annotated literals $(l_1^*, \ldots, l_n^*)$, where $l_i^* \in \{l^{\text{dec}}, l^\omega\}$, that records the order of extensions, together with the reason for each extension, either as a decision or as a clause. Each entry in the trail is of the form $l^{\text{dec}}$ if the literal $l$ evaluates to 1 due to a decision, or $l^\omega$ if $l$ evaluates to 1 due to a unit propagation step from a clause $\omega$.

The value assigned to each variable can be obtained by following the trail, since for each annotated literal $l^* \in \{l^{\text{dec}}, l^\omega\}$ the variable $x$ such that $l \in \{x, \neg x\}$ is assigned a value so that $l^\tau = 1$ holds. The decision level of an annotated literal is defined as the number of decision literals $l^{\text{dec}}$ that appear before or at that point in the trail.

**Example.** Let $\varphi = \{\{\neg x_1, \neg x_2\}, \{x_2, x_3\}, \{\neg x_3\}\}$, and let $\tau$ be an assignment such that $\tau(x_1) = \tau(x_2) = \tau(x_3) = u$.

Since no clause is unit under $\tau$, we cannot perform unit propagation. We make the decision to extend $\tau$ with $\tau(x_1) := 1$. Now we can perform unit propagation, considering the clauses from left to right.

The first clause $\{\neg x_1, \neg x_2\}$ becomes unit, since $(\neg x_1)^\tau = 0$ and $(\neg x_2)^\tau = u$. Thus, $x_2$ is implied and $\tau$ is extended with $\tau(x_2) := 0$.

Next, the clause $\{x_2, x_3\}$ becomes unit because $x_2^\tau = 0$ and $x_3^\tau = u$. Therefore, $x_3$ is implied and $\tau$ is extended with $\tau(x_3) := 1$.

Finally, the clause $\{\neg x_3\}$ is falsified, since $(\neg x_3)^\tau = 0$, and a conflict occurs.

The resulting trail is

$$x_1^{\text{dec}}, \quad \neg x_2^{\{\neg x_1, \neg x_2\}}, \quad x_3^{\{x_2, x_3\}}.$$

The corresponding assignment is $\tau = \{x_1 \mapsto 1,\ x_2 \mapsto 0,\ x_3 \mapsto 1\}$, and the decision level of all literals is 1.

Trails can be represented as acyclic directed graphs, called implication graphs. A directed graph is a pair $(V, E)$, where $V$ is a set of vertices and $E \subseteq V \times V$ is a set of directed edges. A path in a directed graph is a sequence of vertices $(v_1, v_2, \ldots, v_k)$ such that $(v_i, v_{i+1}) \in E$ for all $i = 1, \ldots, k-1$. The graph is acyclic if there is no non-empty path from any vertex back to itself.

Given a trail $\pi$, the corresponding implication graph $G_\pi = (V, E)$ is constructed as follows. Each vertex in $V$ corresponds to a literal that appears in $\pi$. For each entry of $\pi$ in the form of $l^\omega$, where $\omega = \{l_1, \ldots, l_k, l\}$ is the clause that implied $l$, the graph contains directed edges $(\neg l_1, l), \ldots, (\neg l_k, l)$. If the assignment falsifies a clause $\{l_1, \ldots, l_k\}$, a special vertex $\bot$ is added to $V$, and the edges $(\neg l_1, \bot), \ldots, (\neg l_k, \bot)$ are included in $E$. By construction, $G_\pi$ is acyclic.

**Example.** Let $\varphi = \{\{\neg x_1, \neg x_2\},\ \{x_2, x_3\},\ \{\neg x_3\}\}$ and suppose the trail after unit propagation is

$$x_1^{\text{dec}},\ \neg x_2^{\{\neg x_1, \neg x_2\}},\ x_3^{\{x_2, x_3\}}.$$

The first entry $x_1^{\text{dec}}$ has no incoming edges. The second entry $\neg x_2^{\{\neg x_1, \neg x_2\}}$ adds an edge $(x_1, \neg x_2)$. The third entry $x_3^{\{x_2, x_3\}}$ adds an edge $(\neg x_2, x_3)$.

Now the clause $\{\neg x_3\}$ is falsified, so it becomes the conflict clause. We add a special vertex $\bot$ and an edge $(x_3, \bot)$. The resulting implication graph has $V = \{x_1, \neg x_2, x_3, \bot\}$ and $E = \{(x_1, \neg x_2),\ (\neg x_2, x_3),\ (x_3, \bot)\}$.

In the CDCL algorithm, conflict analysis examines a conflict and derives a new clause, called a learned clause, that prevents it from reoccurring. This process is best described using the implication graph. A partition of the vertex set $V$ of $G_\pi$ is a pair of disjoint sets $(A, B)$ such that $A \cup B = V$. From an implication graph $G_\pi$ obtained from a trail that falsifies at least one clause, we can identify the reason for the conflict by selecting a partition $(A, B)$ in which all decision literal vertices belong to $A$ and $\bot$ belongs to $B$. Such a partition is called a conflict cut. Let

$$R = \{l \in A \mid \exists\, l' \in B : (l, l') \in E\}$$

be the reason set of the cut, consisting of the vertices in $A$ with edges to $B$. The learned clause corresponding to the cut is

$$\bigvee_{l \in R} \neg l.$$

If all literals in $R$ evaluate to 1 under an assignment, the learned clause is falsified and the same conflict occurs. Adding the learned clause to the formula preserves all satisfying assignments and prevents the same conflict from arising again.

**Example.** Let $\varphi = \{\{\neg x_1, \neg x_2\},\ \{x_2, x_3\},\ \{\neg x_3, x_4\},\ \{\neg x_4\}\}$ and suppose the trail is

$$x_1^{\text{dec}}, \quad \neg x_2^{\{\neg x_1, \neg x_2\}}, \quad x_3^{\{x_2, x_3\}}, \quad x_4^{\{\neg x_3, x_4\}}$$

with conflict clause $\{\neg x_4\}$. The implication graph contains vertices $x_1$, $\neg x_2$, $x_3$, $x_4$, and $\bot$, and edges $(x_1, \neg x_2)$, $(\neg x_2, x_3)$, $(\neg x_3, x_4)$, $(x_4, \bot)$.

One possible conflict cut is

$$A = \{x_1, \neg x_2, x_3\}, \quad B = \{x_4, \bot\},$$

giving

$$R = \{x_3\}.$$

The learned clause corresponding to this conflict cut is $\{\neg x_3\}$. If $x_3$ is assigned 1, unit propagation implies $x_4$ and immediately falsifies the clause $\{\neg x_4\}$, reproducing the conflict.

A vertex $l$ in an implication graph $G_\pi$ obtained from a trail that falsifies at least one clause is called a unique implication point (UIP) if all paths from the most recent decision literal to $\bot$ go through $l$. The most recent decision literal is always a UIP by definition. Among all UIPs, the first UIP is the one closest to the conflict vertex along any path from the latest decision literal. A UIP cut is a conflict cut where $B$ consists of the UIP and all vertices reachable from it, and $A$ contains the rest. The conflict cut in the previous example is, for example, such a first UIP cut. The learned clause from a first UIP cut contains exactly one literal from the most recent decision level.

After learning a clause $\omega$ from a first UIP cut, the CDCL algorithm backjumps to the highest decision level $m$ among the literals of $\omega$ that is strictly less than the current decision level. A backjump is the removal of all literals from the trail whose decision level is greater than $m$, thereby discarding every assignment that depended on more recent decisions. Because $\omega$ contains exactly one literal from the latest decision level, after backjumping it becomes unit under the assignment and immediately implies that literal.

## 2.3 DIMACS Format

SAT solvers commonly take input in the DIMACS format, which represents a CNF formula as plain text. The file may begin with comment lines starting with the letter $c$. These are followed by a problem line of the form $p$ cnf $v$ $c$, where $v$ is the number of variables and $c$ is the number of clauses.

Each clause is written as a sequence of non-zero integers followed by 0. A positive integer $i$ represents the literal $x_i$, and a negative integer $-i$ represents the literal $\neg x_i$. Clauses may span multiple lines. The order of literals within a clause and the order of clauses in the file do not matter.

For example, the CNF formula $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_4 \vee \neg x_1)$ is represented in DIMACS format as

```
c Example CNF
p cnf 4 2
1 -2 0
3 4 -1 0
```

# 3 Nonogram Encodings

In order to apply SAT solvers to Nonogram puzzles, we require encodings that translate the puzzle constraints into a formula. We first introduce Nonograms formally. We then describe how such puzzles can be represented as CNF formulas by introducing Boolean variables and clauses that capture the constraints given by the clues.

## 3.1 Nonograms

A Nonogram is a puzzle defined on a grid of $n \times m$ cells, where each cell must be colored either black or white. A line $L_i$ refers to either the $i$-th row $R_i$ or the $i$-th column $C_i$ of the grid. The length $l \in \{n, m\}$ of line $L_i$ is given by the number of cells in it. A coloring of a line $L_i$ of length $l$ is a sequence $p \in \{0, 1\}^l$, where 0 denotes a white cell and 1 denotes a black cell.

The puzzle is specified by $n$ row clues and $m$ column clues. Each line $L_i$ is associated with a clue $(g_1, g_2, \ldots, g_t)$, where $g_j \in N^+$ for $j = 1, \ldots, t$, and each $g_j$ specifies a block of $g_j$ consecutive black cells in the line. The blocks must appear in the given order and be separated by at least one white cell. Additional white cells may occur before the first block and after the last block. A coloring satisfies a clue $(g_1, \ldots, g_t)$ if its black cells form exactly $t$ maximal consecutive blocks, and the lengths of these blocks are $g_1, \ldots, g_t$ in this order.

We consider only Nonograms that have a unique solution. A solution for a Nonogram is a coloring of all lines such that every clue is satisfied. A solution for a line is a coloring that satisfies its clue. We note that a single line may have multiple solutions.

## 3.2 Encodings

An encoding is a representation of the Nonogram as a CNF formula $\varphi$ such that any assignment found by a SAT solver corresponds exactly to a solution of the puzzle.

The construction of the encoding starts with an empty set of Boolean variables $X$ and an empty formula $\varphi$. Variables are added to represent structural properties of the puzzle, such as a specific cell being black or a block starting at a certain position. Assigning the value 1 to a variable means that the property it represents holds in the grid.

Each clause in $\varphi$ represents part of the conditions that together enforce the clues of the puzzle. If every clause in $\varphi$ evaluates to 1, then all clues are satisfied.

Many clauses arise from the fact that in a solution exactly one of several structural properties can hold. This is typically encoded by combining an at-least-one clause with multiple at-most-one clauses.

Let $L = \{l_1, \ldots, l_k\}$ be a finite set of literals. The at-least-one clause is

$$(l_1 \vee l_2 \vee \cdots \vee l_k),$$

ensuring that at least one literal evaluates to 1.

The at-most-one clauses are

$$\bigwedge_{1 \le i < j \le k} (\neg l_i \vee \neg l_j),$$

ensuring that at most one literal evaluates to 1. This encoding introduces $\binom{k}{2}$ clauses [4]. An encoding that introduces less clauses uses auxiliary variables $s_1, \ldots, s_{k-1}$ and the

sequential counter construction. The clauses are

$$\bigwedge_{i=1}^{k-1} (\neg l_i \vee s_i) \wedge (\neg s_i \vee s_{i+1}) \wedge (\neg s_i \vee \neg l_{i+1}),$$

which ensure that if $l_i$ evaluates to 1, then all $l_j$ with $j > i$ do not evaluate to 1. This encoding requires $k - 1$ auxiliary variables and $3k - 4$ clauses [4].

## 3.3 Approach

In this thesis we introduce two SAT encodings for Nonograms: the Sequence Enumeration encoding and the Block-Based encoding. Both encodings are translations of the CSP-based approaches of Aramian and Yeghiazaryan [2] into CNF formulas.

In the following sections we first describe each encoding in detail. We then compare how they relate to the CSP approaches on which they are based. Then, in the experimental evaluation, we will compare the two SAT encodings against each other.

# 4 Sequence Enumeration

In this section we introduce the Sequence Enumeration encoding. For each line we define the set of all of its solutions. We then represent each solution by a Boolean variable. Based on these variables we then describe how to construct an encoding of the Nonogram.

## 4.1 Prerequisites

We assume an $n \times m$ Nonogram, an empty formula $\varphi$, and an empty variable set $X$. A solution of a line $L_i$ of length $l$ is a coloring that satisfies the clue of $L_i$. The set of all such solutions for $L_i$ is

$$S_1^{L_i} = \{\, s \in \{0,1\}^l \mid s \text{ satisfies the clue of } L_i \}.$$

We distinguish between two types of lines. Either all rows are of the first and all columns are of the second type, or all columns are of the first and all rows are of the second type. For a line of the first type $L_i^1$, every solution in $S_1^{L_i^1}$ is represented by a Boolean variable. For a line of the second type $L_j^2$, only those solutions are represented that are consistent with the solutions of the first type. Two solutions are consistent if they belong to intersecting lines and assign the same color to the intersecting cell. The set of represented solutions of $L_j^2$ is

$$S_2^{L_j^2} = \{\, s \in S_1^{L_j^2} \mid s \text{ is consistent with some } t \in S_1^{L_i^1},\ i = 1, \ldots, l \}.$$

We aim to generate as few solutions as possible. Therefore, we must consider whether it is more efficient to generate $S_2^{C_j}$ from $S_1^{R_i}$ or $S_2^{R_i}$ from $S_1^{C_j}$. We assume that the ratios

$$\frac{|S_2^{C_j}|}{|S_1^{C_j}|} \quad \text{and} \quad \frac{|S_2^{R_i}|}{|S_1^{R_i}|}$$

are similar. It is then advantageous to start with the type of lines that have fewer solutions in $S_1$, since this excludes more solutions when generating $S_2$ for the other type of line.

To count the sizes of the sets $S_1^{R_1}, \ldots, S_1^{R_n}, S_1^{C_1}, \ldots, S_1^{C_m}$ without explicitly generating the solutions, we use the following result.

**Theorem (Stars and Bars) [15, Section 1.2].** For positive integers $n$ and $k$, the number of $k$-tuples of non-negative integers whose sum is $n$ is

$$\binom{n+k-1}{k-1}.$$

Consider a line $L_i$ of length $l$ with clue $(g_1, g_2, \ldots, g_t)$. Let $b = g_1 + g_2 + \cdots + g_t$ be the total number of black cells, so the number of white cells is $l - b$. The clue requires that $t$ blocks are placed in the line with at least one white cell between blocks.

We introduce a $(t+1)$-tuple $(x_1, x_2, \ldots, x_{t+1})$ of non-negative integers, where $x_1$ is the number of white cells before the first block, $x_{t+1}$ after the last block, and $x_i$ represents the number of white cells between the $i-1$-th and $i$-th block for $i \in \{2, \ldots, t\}$. Since $t-1$ white cells are reserved as mandatory separators, the remaining $l - b - (t-1)$ white

cells can be freely distributed among the $t+1$ positions. By the stars and bars theorem, with $n = l - b - (t-1)$ and $k = t+1$, the number of possible tuples is

$$\binom{l-b+1}{t}.$$

Each tuple corresponds to exactly one solution for the line, so

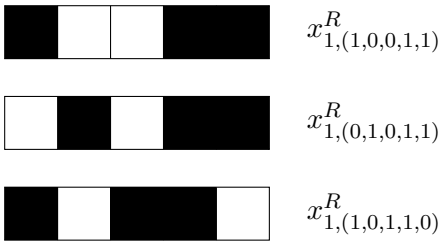
$$|S_1^{L_i}| = \binom{l-b+1}{t}.$$

## 4.2 Encoding

The construction of the encoding starts by adding, for each line $L_i^1$ and each solution $s \in S_1^{L_i^1}$, the Boolean variable $x_{i,s}^{L^1}$ to $X$. For each line $L_j^2$ and each solution $t \in S_2^{L_j^2}$, we add the Boolean variable $x_{j,t}^{L^2}$ to $X$. We call these solution variables. Assigning the value 1 to a solution variable means that the line takes the corresponding solution.

**Example.** Consider the first row of the Nonogram in Figure 1:


For this row, there are three solutions. Thus, we add the solution variables:




To enforce that each line has exactly one solution, we first add at-least-one clauses and then at-most-one clauses to $\varphi$. The at-least-one clauses are

$$\bigvee_{s \in S_1^{L_i^1}} x_{i,s}^{L^1} \quad \text{for each line } L_i^1,$$

$$\bigvee_{t \in S_2^{L_j^2}} x_{j,t}^{L^2} \quad \text{for each line } L_j^2.$$

Let $k = |S_1^{L_i^1}|$ be the number of solutions for line $L_i^1$ and $p = |S_2^{L_j^2}|$ the number of solutions for line $L_j^2$. We add to $X$ the auxiliary Boolean variables $u_{i,1}^{L^1}, \ldots, u_{i,k-1}^{L^1}$ for $L_i^1$ and $u_{j,1}^{L^2}, \ldots, u_{j,p-1}^{L^2}$ for $L_j^2$. The at-most-one clauses are

$$\bigwedge_{r=1}^{k-1} (\neg x_{i,s_r}^{L^1} \vee u_{i,r}^{L^1}) \wedge (\neg u_{i,r}^{L^1} \vee u_{i,r+1}^{L^1}) \wedge (\neg u_{i,r}^{L^1} \vee \neg x_{i,s_{r+1}}^{L^1}) \quad \text{for each line } L_i^1,$$

$$\bigwedge_{r=1}^{p-1} (\neg x^{L^2}_{j,t_r} \vee u^{L^2}_{j,r}) \wedge (\neg u^{L^2}_{j,r} \vee u^{L^2}_{j,r+1}) \wedge (\neg u^{L^2}_{j,r} \vee \neg x^{L^2}_{j,t_{r+1}}) \quad \text{for each line } L^2_j.$$

The coloring of two intersecting lines must have the same color in their intersection cell for it to be a solution for the Nonogram. To encode this, we add consistency clauses to $\varphi$. First, we add the Boolean variables $b_{i,j}$ for every cell $(i, j)$ to $X$. We call them cell variables. Assigning 1 to a cell variable means that the corresponding cell in the Nonogram is colored black. Without loss of generality, we assume that all rows are of the first type and all columns are of the second type.

Then, for each solution $s \in S_1^{R_i}$ the consistency clauses are

$$\neg x^R_{i,s} \vee b_{i,j} \quad \text{if the } j\text{-th entry of } s \text{ is 1,}$$

$$\neg x^R_{i,s} \vee \neg b_{i,j} \quad \text{if the } j\text{-th entry of } s \text{ is 0.}$$

For each solution $t \in S_2^{C_j}$ the consistency clauses are

$$\neg x^C_{j,t} \vee b_{i,j} \quad \text{if the } i\text{-th entry of } t \text{ is 1,}$$

$$\neg x^C_{j,t} \vee \neg b_{i,j} \quad \text{if the } i\text{-th entry of } t \text{ is 0.}$$

**Example.** Suppose $s \in S_1^{R_i}$ represents a solution in which cell $(i, j)$ is black and $t \in S_2^{C_j}$ represents a solution in which the same cell is white. Then we add

$$\neg x^R_{i,s} \vee b_{i,j}, \quad \neg x^C_{j,t} \vee \neg b_{i,j}.$$

If both $x^R_{i,s}$ and $x^C_{j,t}$ are assigned the value 1, the first clause enforces $b_{i,j} = 1$ while the second enforces $b_{i,j} = 0$, which is impossible. Therefore $x^R_{i,s}$ and $x^C_{j,t}$ cannot both be assigned the value 1.

## 4.3 Comparison to the CSP Approach

In the CSP approach of Aramian and Yeghiazaryan [2], each row and column is modeled as a non-Boolean variable whose domain is the set of all solutions for that line. Solving proceeds by constraint propagation, where the domains are iteratively reduced by eliminating solutions that are inconsistent with the domains of intersecting lines, until a complete solution is obtained.

In our SAT encoding the role of these domains is taken by the solution set of each line. Each element of a solution set is represented by a Boolean variable, and when constructing the solution sets of the second type of lines we restrict them to only those solutions that are consistent with the already generated first type. This corresponds to the solving method of the CSP approach, but in our case it is applied only once during preprocessing. From that point onward the solution sets remain fixed, and the Boolean variables together with the corresponding constraints are turned into a CNF formula that is then given to the SAT solver.

## 4.4 Proof of Correctness

**Theorem (Sequence Enumeration).** The Sequence Enumeration Encoding is satisfiable if and only if the Nonogram has a solution.

*Proof.* "⇒" Assume that the CNF formula $\varphi$ constructed by the Sequence Enumeration encoding is satisfiable. Then there exists an assignment $\tau$ that satisfies all clauses in $\varphi$.

For each line $L_i^1$, the at-least-one clause

$$\bigvee_{s \in S_1^{L_i^1}} x_{i,s}^{L^1}$$

ensures that at least one solution variable $x_{i,s}^{L^1}$ is assigned the value 1, and the at-most-one clauses enforce that no two such variables are assigned the value 1. Similarly, for each line $L_j^2$, the at-least-one clause

$$\bigvee_{t \in S_2^{L_j^2}} x_{j,t}^{L^2}$$

together with its at-most-one clauses ensures that exactly one solution variable $x_{j,t}^{L^2}$ is assigned the value 1. Therefore, $\tau$ assigns exactly one solution variable the value 1 for every line in the Nonogram.

For each cell $(i, j)$, the consistency clauses

$$\neg x_{i,s}^{L^1} \vee b_{i,j}, \quad \neg x_{i,s}^{L^1} \vee \neg b_{i,j}, \quad \neg x_{j,t}^{L^2} \vee b_{i,j}, \quad \neg x_{j,t}^{L^2} \vee \neg b_{i,j}$$

ensure that, without loss of generality, if $x_{i,s}^{L^1}$ represents the selected solution of row $i$ and $x_{j,t}^{L^2}$ represents the selected solution of column $j$, and both are assigned the value 1, then the coloring of cell $(i, j)$ in $s$ and $t$ must agree. Since $\tau$ satisfies all consistency clauses, the colors represented by the selected row and column solutions agree on every cell variable $b_{i,j}$.

By construction, every solution $s \in S_1^{L_i^1}$ and $t \in S_2^{L_j^2}$ represents a coloring that satisfies the clue of its corresponding line. Therefore, $\tau$ represents a coloring of the entire grid in which every clue is satisfied and all cells are consistent. Thus, $\tau$ corresponds to a solution of the Nonogram.

"⇐" Assume the Nonogram has a solution. Then there exists a coloring of the $n \times m$ grid that satisfies all row and column clues.

For each line $L_i^1$, let $s \in S_1^{L_i^1}$ be the solution of $L_i^1$. Assign the value 1 to $x_{i,s}^{L^1}$ and assign the value 0 to $x_{i,s'}^{L^1}$ for all $s' \in S_1^{L_i^1}$, $s' \neq s$. For each line $L_j^2$, let $t \in S_2^{L_j^2}$ be the solution of $L_j^2$. Assign the value 1 to $x_{j,t}^{L^2}$ and assign the value 0 to $x_{j,t'}^{L^2}$ for all $t' \neq t$.

For each cell $(i, j)$, assign the value 1 to $b_{i,j}$ if the cell is black in the solution and assign the value 0 otherwise.

Under this assignment, each line has exactly one solution variable assigned the value 1, satisfying all at-least-one and at-most-one clauses. The consistency clauses are satisfied because the selected row and column solutions agree on the color of every cell. Hence, $\tau$ satisfies all clauses in $\varphi$. □

## 4.5 Variable and Clause Counts

Let

$$L^1 = \sum_{i=1}^{n} |S_1^{L_i^1}|, \quad L^2 = \sum_{j=1}^{m} |S_2^{L_j^2}|.$$

We introduce $L^1 + L^2$ solution variables and $nm$ cell variables. The at-most-one encoding for each line $L_i^1$ with $k = |S_1^{L_i^1}|$ adds $k-1$ auxiliary variables, and for each line $L_j^2$ with $p = |S_2^{L_j^2}|$ adds $p-1$ auxiliary variables. In total, this is $(L^1 - n) + (L^2 - m)$ auxiliary variables. Hence, the total number of variables is

$$(L^1 + L^2) + nm + (L^1 - n) + (L^2 - m) = 2L^1 + 2L^2 + nm - (n+m).$$

The total number of clauses is the sum of $n+m$ at-least-one clauses, $\big(3(L^1+L^2) - 4(n+m)\big)$ at-most-one clauses, and $mL^1 + nL^2$ consistency clauses. Therefore, the total number of clauses is

$$(n+m) + \big(3(L^1+L^2) - 4(n+m)\big) + (mL^1 + nL^2) = mL^1 + nL^2 + 3(L^1+L^2) - 3(n+m).$$

Using the result

$$|S_1^{L_i}| = \binom{l-b+1}{k},$$

where $l$ is the line length, $b$ is the number of black cells in the line, and $k$ is the number of blocks, $|S_1^{L_i}|$ is the largest when all blocks have length 1 [2], giving $b = k$ and

$$|S_1^{L_i}| = \binom{l-k+1}{k}.$$

This binomial coefficient is maximized when $k = \lfloor l/3 \rfloor$, giving

$$|S_1^{L_i}|_{\max} = \binom{l - \lfloor l/3 \rfloor + 1}{\lfloor l/3 \rfloor}.$$

Then in the worst case

$$L^1 = n\binom{m - \lfloor m/3 \rfloor + 1}{\lfloor m/3 \rfloor}, \quad L^2 = m\binom{n - \lfloor n/3 \rfloor + 1}{\lfloor n/3 \rfloor}.$$

Assuming a $10 \times 10$ Nonogram, the worst case has $k = \lfloor 10/3 \rfloor = 3$ and

$$|S_1^{L_i}|_{\max} = \binom{10-3+1}{3} = \binom{8}{3} = 56,$$

so $L^1 = L^2 = 10 \cdot 56 = 560$. Then the total number of variables is

$$2L^1 + 2L^2 + nm - (n+m) = 2 \cdot 560 + 2 \cdot 560 + 100 - 20 = 2320,$$

and the total number of clauses is

$$mL^1 + nL^2 + 3(L^1+L^2) - 3(n+m) = 10 \cdot 560 + 10 \cdot 560 + 3(560+560) - 30 = 14500.$$

# 5 Block-Based Encoding

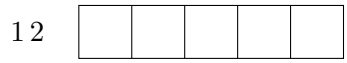
In this section we introduce the Block-Based encoding. For each block in a line, we define the set of all start positions that this block can take based on the solutions of that line. We then represent each start position by a Boolean variable. Based on these variables we then describe how to construct an encoding of the Nonogram.

## 5.1 Encoding

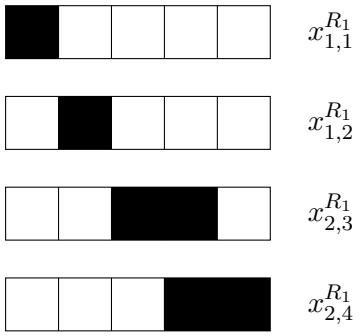
We assume an $n \times m$ Nonogram with an empty formula $\varphi$ and an empty set of variables $X$. Consider a line $L_i$ of length $l$ with clue $(g_1, \ldots, g_k)$. For each block $b \in \{1, \ldots, k\}$ and each start position $s \in \{1, \ldots, l\}$, if there exists a solution of $L_i$ in which block $b$ begins at position $s$, we add a Boolean variable $x_{b,s}^{L_i}$ to $X$. We call these variables block variables. Assigning 1 to $x_{b,s}^{L_i}$ means that block $b$ begins at position $s$ in line $L_i$. The set of all such start positions for block $b$ is

$$S_b^{L_i} = \{\, s \in \{1, \ldots, l\} \mid \text{there is a solution of } L_i \text{ in which block } b \text{ begins at } s\}.$$

**Example.** Consider the first row of the Nonogram in Figure 1




Block 1 and 2 both have two start positions that satisfy the clue. Therefore we add four block variables




We encode that in a solution for the Nonogram each block in each line has exactly one start position by first adding at-least-one clauses and then at-most-one clauses to $\varphi$. The at-least-one clauses are

$$\bigvee_{s \in S_b^{L_i}} x_{b,s}^{L_i} \quad \text{for each line } L_i \text{ and block } b.$$

Let $t = |S_b^{L_i}|$ be the number of start positions for block $b$ in line $L_i$. We add to $X$ the auxiliary Boolean variables $u_{b,1}^{L_i}, \ldots, u_{b,t-1}^{L_i}$ for each line $L_i$ and block $b$. The at-most-one clauses are

$$\bigwedge_{j=1}^{t-1} (\neg x_{b,s_{b,j}^{L_i}}^{L_i} \vee u_{b,j}^{L_i}) \wedge (\neg u_{b,j}^{L_i} \vee u_{b,j+1}^{L_i}) \wedge (\neg u_{b,j}^{L_i} \vee \neg x_{b,s_{b,j+1}^{L_i}}^{L_i}) \quad \text{for each line } L_i \text{ and block } b.$$

Two consecutive blocks in the same line must have at least one white cell between them. To encode this, for each line $L_i$, each pair of consecutive blocks $b$ and $b+1$, and each pair of start positions $s \in S_b^{L_i}$ and $t \in S_{b+1}^{L_i}$ such that $s + g_b \geq t$ we add a block consistency clause. The block consistency clause is

$$\neg x_{b,s}^{L_i} \vee \neg x_{b+1,t}^{L_i}.$$

In a solution for a Nonogram, a cell $(i, j)$ must have the same coloring in both of its intersecting lines. To encode this, we add cell consistency clauses.

For line $L_i$ and a cell index $p$, we define

$$B_p^{L_i} = \{\, x_{b,s}^{L_i} \in X \mid s \leq p < s + g_b \}$$

as the set of block variables whose start positions represent that cell $p$ in line $L_i$ is black.

For each $x \in B_j^{R_i}$ and each $y \in B_i^{C_j}$, the cell consistency clauses are

$$\neg x \vee \bigvee_{y' \in B_i^{C_j}} y', \quad \neg y \vee \bigvee_{x' \in B_j^{R_i}} x'.$$

**Example.** Suppose $B_j^{R_i} = \{x_1, x_2\}$ and $B_i^{C_j} = \{y_1\}$. The cell consistency clauses are

$$\neg x_1 \vee y_1, \quad \neg x_2 \vee y_1, \quad \neg y_1 \vee x_1 \vee x_2.$$

If $x_1$ is assigned the value 1, then $\neg x_1 \vee y_1$ implies $y_1$. Similarly, if $y_1$ is assigned the value 1, then $\neg y_1 \vee x_1 \vee x_2$ implies that at least one of $x_1$ or $x_2$ is assigned the value 1. Thus the row and column agree on the cell being black.

## 5.2 Comparison to the CSP Approach

In the CSP approach of Aramian and Yeghiazaryan [2], each row and column is represented as a CSP whose variables are the start positions of the blocks. The domain of a variable is the set of all possible start positions for that block, and constraints ensure that consecutive blocks are separated by at least one white cell. Solving proceeds by systematically enumerating all line solutions using backtracking.

In our Block-Based SAT encoding, the same domains are instead represented as solution sets containing only those start positions that occur in at least one line solution. Each start position is turned into a Boolean variable, and clauses enforce that consecutive blocks are separated. Unlike the CSP approach, where consistency between rows and columns is maintained by the solving procedure, in our encoding we must explicitly add consistency clauses to ensure that intersecting rows and columns agree on cells. For solving, the Boolean variables and constraints are encoded into a CNF formula, which is then passed to the SAT solver.

## 5.3 Proof of Correctness

**Theorem (Block-Based Encoding).** The Block-Based Encoding is satisfiable if and only if the Nonogram has a solution.

*Proof.* "⇒" Assume that the CNF formula $\varphi$ constructed by the Block-Based encoding is satisfiable. Then there exists an assignment $\tau$ that satisfies all clauses in $\varphi$.

For each line $L_i$ and each block $b \in \{1, \ldots, k\}$, the at-least-one clause

$$\bigvee_{s \in S_b^{L_i}} x_{b,s}^{L_i}$$

ensures that at least one block variable $x_{b,s}^{L_i}$ is assigned the value 1. The at-most-one clauses enforce that no two distinct block variables for the same block are assigned the value 1. Therefore, exactly one start position is chosen for each block in every line.

For each line $L_i$, each pair of consecutive blocks $b$ and $b+1$, and each pair of start positions $s \in S_b^{L_i}$ and $t \in S_{b+1}^{L_i}$ such that $s + g_b \geq t$, the block consistency clause

$$\neg x_{b,s}^{L_i} \vee \neg x_{b+1,t}^{L_i}$$

prevents the two blocks from overlapping or touching, ensuring that there is at least one white cell between them.

For each cell $(i, j)$, the cell consistency clauses

$$\neg x \vee \bigvee_{y \in B_i^{C_j}} y, \quad \neg y \vee \bigvee_{x \in B_j^{R_i}} x$$

where $x \in B_j^{R_i}$ and $y \in B_i^{C_j}$, ensure that if a row block variable implies that cell $(i, j)$ is black, then at least one column block variable does as well, and vice versa. Since $\tau$ satisfies all cell consistency clauses, every cell is assigned the same color in both orientations.

By construction, every start position $s \in S_b^{L_i}$ is derived from a solution of $L_i$. Therefore, $\tau$ corresponds to a solution of the Nonogram.

"⇐" Assume the Nonogram has a solution. Then there exists a coloring of the $n \times m$ grid that satisfies all row and column clues.

For each line $L_i$ and each block $b \in \{1, \ldots, k\}$, let $s \in S_b^{L_i}$ be the start position of block $b$ in the solution. Assign the value 1 to $x_{b,s}^{L_i}$ and assign the value 0 to $x_{b,s'}^{L_i}$ for all $s' \in S_b^{L_i}$ with $s' \neq s$.

Under this assignment, the at-least-one clauses are satisfied because each block has exactly one chosen start position, and the at-most-one clauses are satisfied because no two start positions for the same block are chosen. The block consistency clauses are satisfied because the blocks in the solution have at least one white cell between them. The cell consistency clauses are satisfied because the solution colors each cell identically from the row and column perspectives.

Hence, the assignment $\tau$ satisfies all clauses in $\varphi$, so $\varphi$ is satisfiable. □

## 5.4 Variable and Clause Counts

Let

$$B = \sum_{i=1}^{n+m} \sum_{b=1}^{k_i} |S_b^{L_i}|$$

be the total number of block variables, where $k_i$ is the number of blocks in line $L_i$, and let

$$T = \sum_{i=1}^{n+m} k_i$$

be the total number of blocks across all lines. The at-most-one encoding for each $S_b^{L_i}$ of size $t$ adds $t-1$ auxiliary variables, giving a total of $B - T$ auxiliary variables. Hence, the total number of variables is

$$B + (B - T) = 2B - T.$$

The total number of clauses is the sum of $T$ at-least-one clauses, $3B - 4T$ at-most-one clauses from the sequential counter encoding, $Q$ block consistency clauses, and $C$ cell consistency clauses, where

$$Q = \sum_{i=1}^{n+m} \sum_{b=1}^{k_i - 1} \big|\{(s,t) \in S_b^{L_i} \times S_{b+1}^{L_i} \mid s + g_b^{L_i} \geq t\}\big|,$$

$$C = \sum_{i=1}^{n} \sum_{j=1}^{m} \big(|B_j^{R_i}| + |B_i^{C_j}|\big).$$

Therefore, the total number of clauses is

$$T + (3B - 4T) + Q + C = 3B - 3T + Q + C.$$

A line $L_i$ of length $l_i$ has at most $\lceil \frac{l_i}{2} \rceil$ blocks, since any larger number would leave no space for the mandatory white cells separating consecutive blocks. Hence

$$k_i \leq \lceil \frac{l_i}{2} \rceil.$$

Furthermore, a single block can have at most $l_i$ start positions, so

$$|S_b^{L_i}| \leq l_i.$$

With this we can give upper bounds for $T, B, Q$ and $C$. The total number of blocks is bounded by

$$T = \sum_{i=1}^{n+m} k_i \leq \frac{1}{2} \sum_{i=1}^{n+m} l_i = \frac{1}{2}(nm + nm) = nm.$$

The total number of block variables is bounded by

$$B = \sum_{i=1}^{n+m} \sum_{b=1}^{k_i} |S_b^{L_i}| \leq \sum_{i=1}^{n+m} k_i \, l_i \leq \sum_{i=1}^{n+m} \frac{l_i}{2} \, l_i = \frac{1}{2} \sum_{i=1}^{n+m} l_i^2 = \frac{1}{2}\big(nm^2 + mn^2\big) = \frac{1}{2} \, nm \, (m+n).$$

For the number of block consistency clauses $Q$, each pair of consecutive blocks in $L_i$ satisfies

$$\left|\{(s,t) \in S_b^{L_i} \times S_{b+1}^{L_i}\}\right| \;\leq\; l_i^2,$$

since each set of start positions has size at most $l_i$. $L_i$ can contain at most $k_i \leq \lceil \frac{l_i}{2} \rceil$ blocks, and thus at most $l_i/2$ consecutive block pairs. Therefore

$$Q \leq \sum_{i=1}^{n+m} \frac{l_i}{2}\, l_i^2 = \frac{1}{2} \sum_{i=1}^{n+m} l_i^3 = \frac{1}{2}\big(nm^3 + mn^3\big) = \frac{1}{2}\, nm\,(m^2+n^2).$$

For the number of cell consistency clauses $C$, we have $|B_j^{R_i}| \leq k_{R_i}$ and $|B_i^{C_j}| \leq k_{C_j}$ for each cell $(i,j)$. Therefore

$$C \leq \sum_{i=1}^{n}\sum_{j=1}^{m}\big(k_{R_i} + k_{C_j}\big) = m\sum_{i=1}^{n} k_{R_i} + n\sum_{j=1}^{m} k_{C_j} \leq m \cdot \frac{nm}{2} + n \cdot \frac{mn}{2} = \frac{1}{2}\, nm\,(m+n).$$

Assuming a $10 \times 10$ Nonogram, we have the bounds

$$T \leq nm = 100, \quad B \leq \tfrac{1}{2}\, nm(m+n) = 1000,$$

$$Q \leq \tfrac{1}{2}\, nm(m^2+n^2) = 10000, \quad C \leq \tfrac{1}{2}\, nm(m+n) = 1000.$$

Then the total number of variables is

$$2B - T \leq 2 \cdot 1000 - 100 = 1900,$$

and the total number of clauses is

$$3B - 3T + Q + C \leq 3 \cdot 1000 - 3 \cdot 100 + 10000 + 1000 = 13700.$$

# 6 CDCL Solver

In this section we describe the implementation of our CDCL solver. The design follows the general descriptions of the CDCL algorithm given in [11, 9], with unit propagation and conflict analysis based on the minimal reference implementation in [5]. We extend this baseline by modifying unit propagation to improve efficiency on binary clauses and by adding decision heuristics, restart strategies aimed at improving performance on Nonograms.

## 6.1 Overview

Our CDCL solver does not take as input a DIMACS file but instead directly works with an instance of our encodings. The instance provides the solver with all clauses and variables. The solver relies on three main structures: the trail, the propagation queue, and the phase array.

The trail records variables in the order they are assigned. Each entry stores the assigned value, the reason clause that implied the assignment or empty if it was chosen by the heuristic, and the decision level at which it occurred. From this information the current assignment can be reconstructed, and the decision level of any literal can be determined.

The propagation queue, as used in [5], stores the negated literal of each freshly assigned variable. The assignment is extended by the *enqueue* operation, where a variable together with its assigned value and reason is appended to the trail.

The phase array stores for each variable the polarity to be used when the solver makes a decision. We set it initially to 1, since such an assignment structurally corresponds to a coloring of the Nonogram. Whenever a variable is assigned, its value is stored in the phase array so that the same polarity can be reused if the variable is decided again later in the search. Remembering the last assignment in this way is based on Rsat [13].

Before the main loop begins, the literals of all unit clauses generated by the encoding are enqueued and their negations pushed to the propagation queue. During clause generation, such unit clauses are collected separately so they can be passed directly to the solver at this stage. They correspond to lines that have only one solution.

After these initial assignments, the solver alternates between unit propagation and variable picking. If the propagation queue is not empty, unit propagation is performed. If a conflict is detected, conflict analysis produces a learned clause and the solver performs backjumping according to it before continuing the search. If no conflict occurs and unassigned variables remain, a new variable is picked according to the decision heuristic and enqueued with the polarity given by the phase array. The process terminates with UNSAT if a conflict occurs at decision level 0, or SAT if all variables are eventually assigned without conflict. The corresponding algorithm can be seen in Figure 2.

```
CDCL(encoding)
initialise trail τ, propagation queue Q, phase array A
for each unit clause {l}
  enqueue(l, no reason)
  Q.push(¬other)
while true :
  if Q not empty :
    ω := unitPropagation(τ, Q)
    if ω is a conflict :
      if decision level = 0 : return UNSAT
      ω_learned := analyzeConflict(ω)
      add ω_learned to the formula
      backjump to an earlier decision level according to ω_learned
      enqueue the implied literal from ω_learned with ω_learned as the reason
      add its negation to Q
      continue
  if all variables are assigned : return SAT
  v := pick unassigned variable
  l := literal of v according to A[v]
  enqueue(l, no reason)
  Q.push(¬other)
```

Figure 2: Main loop of our CDCL solver.

## 6.2 Unit Propagation

One way of implementing unit propagation is to repeatedly scan the entire formula. In each step, every clause is examined and the numbers of literals that evaluate to 0 or to $u$ are counted. If all literals evaluate to 0, the clause is falsified and a conflict is detected. If exactly one literal evaluates to $u$ and all others to 0, the clause is unit and the variable of that literal is assigned so that the literal evaluates to 1, with the clause recorded as its reason. The search then restarts from the beginning of the clause list. This continues until no further unit clauses are found. Although simple to implement, this approach is inefficient because it inspects all clauses on every unit propagation step, even when most are unaffected by recent assignments. The corresponding algorithm is shown in Figure 3.

```
unitPropagation(formula φ, trail τ)
while true :
  changed := false
  for each clause  ω ∈ φ :
    count literals in ω evaluating to 0 and u
    if all literals evaluate to 0 :
      return conflict ω
    if exactly one literal evaluates to u :
      assign its variable so that it evaluates to 1
      changed := true
  if not changed :
    break
return no conflict
```

Figure 3: Unit propagation by clause scanning.

Our solver uses a more efficient variant based on the two-watched-literals [12]. Each clause maintains exactly two watched literals. The clause only needs to be inspected when one of its watched literals evaluates to 0, since only then can it become unit or be falsified. If both watched literals evaluate to 1, the clause is already satisfied. If one watched literal evaluates to $u$ and the other does not evaluate to 0, then at least two literals in the clause do not evaluate to 0, and so the clause cannot be unit.

When a watched literal evaluates to 0, the clause is examined to find another literal that does not evaluate to 0. If such a literal exists, it replaces the watched literal that became 0, and the watch lists are updated accordingly. This preserves the invariant that at least one watched literal does not evaluate to 0, which means the clause is neither unit nor falsified. If no replacement exists, then all non-watched literals evaluate to 0, and the status of the clause is determined by the remaining watched literal. If it evaluates to $u$, the clause is unit and the remaining watched literal is implied by assigning its variable with the clause as reason so that the literal evaluates to 1. If it evaluates to 0, then both watched literals are 0 and every non-watched literal is 0 as well, and the clause is falsified, yielding a conflict.

The watch list is indexed by literal and stores the clauses currently watching that literal. Moving a watch removes the clause from the list of the old watched literal and adds it to the list of the new watched literal.

Our solver also implements a binary fast path based on a description by Ryan [14]. Since our formulas contain a large number of binary at-most-one clauses, these are handled separately from the general two-watched-literals scheme. Each literal $l$ maintains two lists: one containing literals and one containing their reason clauses. For every binary clause $\omega = (l \vee r)$ we store two entries. In the list of $\neg l$ we store the literal $r$ together with the clause, and in the list of $\neg r$ we store the literal $l$ together with the clause. During unit propagation, whenever a literal $l$ evaluates to 1, we traverse the list of $\neg l$. If a stored literal $r$ evaluates to 1 nothing is done. If $r$ evaluates to 0 we report a conflict with the stored clause as reason. If $r$ evaluates to $u$ we assign its variable so that $r$ becomes 1, record the clause as its reason, and push $\neg r$ to the propagation queue.

Additionally, we added a simplified version of the blocking literals described in [3]. A blocked literal is a cached literal per clause that, if it evaluates to 1, allows the solver to skip inspecting the clause. In our simplified version the blocked literal is always one of the two watched literals. On clause creation it is set to the second watch. During unit

propagation, if the other watch evaluates to 1 we update the blocker to that watch and stop inspecting the clause. When we move a watch to a literal that evaluates to not 0 we set the blocker to that literal, which then becomes a watch. During unit propagation we first check the blocker. If it evaluates to 1 the clause is already satisfied and we skip any further inspection.

During unit propagation we repeatedly pop a literal $l$ from the propagation queue, corresponding to the negated literal of a newly assigned variable. If binary lists exist for $\neg l$, we traverse them: if the stored literal evaluates to 1 we skip, if it evaluates to 0 we return a conflict, and if it evaluates to $u$ we enqueue it with the clause as reason and push its negation to the queue. After processing the binary lists we handle the clauses that watch $l$. For each clause we first check its blocked literal. If it evaluates to 1 we skip the clause. If the other watched literal evaluates to 1 we update the blocker and skip. Otherwise we attempt to move the watch on $l$ to a literal that evaluates to not 0. If no replacement is found the other watched literal decides the clause: if it evaluates to 0 we return a conflict, and if it evaluates to $u$ we enqueue it and push its negation to the queue. The corresponding algorithm is shown in Figure 4.

```
unitPropagation(trail τ, propagation queue Q)
while Q is not empty :
  l := Q.pop()
  if binary lists exist for ¬l :
    for each entry (r, ω) in lists[¬l] :
      if τ.contains(r) :
        if τ.evaluate(r) = 1 : continue
        return conflict ω
      τ.enqueue(r, ω)
      Q.push(¬r)
  for each clause ω ∈ watches[l] :
    w0 := ω.watch1, w1 := ω.watch2
    other := the watched literal different from l
    if τ.contains(ω.blocker) and τ.evaluate(ω.blocker) = 1 :
      continue
    if τ.contains(other) and τ.evaluate(other) = 1 :
      ω.blocker := other
      continue
    moved := false
    for each literal a in ω except w0, w1 :
      if τ.contains(a) and τ.evaluate(a) = 0 : continue
      move the watch on l to a in ω
      ω.blocker := a
      update watch lists to watch a
      moved := true
      break
    if not moved :
      if τ.contains(other) :
        if τ.evaluate(other) = 0 : return conflict ω
        else :
        τ.enqueue(other, ω)
        Q.push(¬other)
return no conflict
```

Figure 4: Unit propagation with two watched literals.

## 6.3 Conflict Analysis

Our conflict analysis follows the first UIP scheme and works directly on the trail and the reason clauses without constructing an implication graph. When a conflict clause $\omega_{\text{conf}}$ is found, we make it the working clause and initialize a boolean array *seen*, indexed by variables, to false. We scan the working clause to mark its variables as seen and count how many of its literals were assigned at the current decision level. We then walk the trail backwards to the most recently assigned literal at the current level whose variable is marked as seen. If that literal has a reason clause, we add the literals of that reason clause to the working clause, mark any newly seen variables, increase the count for literals at the current level, and collect literals from lower levels while tracking the maximum of their decision levels as the future backjump level. If the literal has no reason we skip it. We repeat the backward walk, each time extending the working clause with the reason clause of the most recent seen literal at the current level, until the counter of current–level

literals becomes zero, so only one literal from the current level remains in the working clause.

At this point the working clause contains only one literal from the current decision level, which is the unique implication point (UIP). This literal is kept in the clause, and the backjump level is determined as the maximum decision level among the other literals. The resulting clause is the learned clause in first UIP form, and the function returns it together with the backjump level. The corresponding algorithm can be seen in Figure 2.

```
analyzeConflict(conflict clause ω_conf, trail τ)
initialize empty list learned
initialize boolean array seen to false
counter := 0, backLevel := 0, ω := ω_conf, idx := τ.size − 1
do :
  for each literal l ∈ ω :
    v := |l|, lvl := τ.decLevel(v)
    if not seen[v] :
      seen[v] := true
      if lvl = τ.level :
        counter := counter + 1
      else if lvl > 0 :
        append l to learned
        if lvl > backLevel : backLevel := lvl
  do :
    p := τ.trail[idx]
    idx := idx − 1
  while not seen[ |p| ]
  ω := τ.reason(|p|)
  counter := counter − 1
while counter > 0
append ¬τ.trail[idx + 1] to learned
return (backLevel, clause built from learned)
```

Figure 5: Conflict analysis.

## 6.4 Decision Heuristics

A decision heuristic determines which unassigned variable the solver selects when it needs to make a decision. We implemented five heuristics: two general-purpose and three tailored to Nonograms. Among the general-purpose heuristics, Variable State Independent Decaying Sum (VSIDS) is widely used in modern SAT solvers. In contrast, our Line-Aware, Block-Length-Aware, and Sequential Order heuristics are designed to exploit structural properties of the Nonogram encodings. Including all five allows us to evaluate which approach is most effective for Nonogram solving.

**Random.** The Random heuristic selects a variable uniformly at random from the set of variables. If the chosen variable is already assigned, a new variable is drawn until a unassigned one is found.

**VSIDS.** In VSIDS [12] each variable is associated with an activity score, initially set to zero. Whenever the literal of a variable appears in a newly learned clause, the variable's activity score is increased, and all scores are periodically decayed so that recent conflicts are weighted more strongly. At each decision point, the unassigned variable with the highest activity score is selected.

In our implementation we avoid explicitly decaying all scores. Instead, we maintain a global increment value that is multiplied by a constant factor after each conflict. Increasing a variable's activity then means adding this increment, which implicitly incorporates the decay over time. This ensures that recent activity is emphasized while keeping the update efficient. This approach is based on [1].

**Line-Aware.** The Line-Aware heuristic uses the fact that each variable belongs to exactly one row or one column of the Nonogram. Whenever a variable is assigned, the corresponding row or column index is added to a queue. When a new decision is required, the solver selects the next unassigned variable from the most recently added line. If the queue is empty, the heuristic falls back to random selection. For each variable the encoder stores in maps its corresponding line and orientation. These maps are then given to the solver, which uses them to enqueue the correct line whenever a variable is assigned.

**Block-Length-Aware.** The Block-Length-Aware heuristic is specific to the Block-Based encoding. The heuristic always selects a currently unassigned variable that belongs to the block with the greatest length. For each block variable the encoder stores in a map its length. This map is then given to the solver.

**Sequential Order.** The Sequential Order heuristic selects unassigned variables in ascending numerical order. In the encoder, variables are numbered consecutively starting from 1. In the Block-Based encoding, all variables of a block are created consecutively, and this order is preserved within each line. Row block variables are generated before column block variables, so decisions proceed line by line from the top left to the bottom right of the grid, and then continue similarly for the columns. In the Sequence Enumeration encoding, either all row sequence variables are generated first or all column sequence variables are generated first. Within each group the variables are ordered line by line, with all sequence variables of one line appearing before those of the next.

## 6.5 Restarts

Restarts in CDCL are a mechanism that interrupts the current search and returns the solver to decision level 0. All assignments on the trail are cleared, except the assignments implied by the initial unit clauses, and the learned clauses remain part of the formula. As shown in [8], this can significantly increase solver performance. For our solver we decided to include two restart strategies, one that produces relatively frequent restarts and one that produces progressively less frequent restarts, in order to test how different restart strategies affect Nonogram solving.

The first is the Luby restart strategy [10]. We define restart intervals according to a recursive sequence $L(i)$, with the interval given by $u \cdot L(i)$. Here $u$ is a fixed unit, which

we set to 32. The Luby sequence is defined for $i \geq 1$ as

$$L(i) = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1, \\ L(i - 2^{k-1} + 1), & \text{if } 2^{k-1} \leq i < 2^k - 1. \end{cases}$$

The first values are

$$1, 1, 2, 1, 1, 2, 4, 1, \ldots$$

With $u = 32$, this means the solver restarts after 32 conflicts, then again after another 32, then after 64, then 32, then 32, then 64, then 128, and so on.

The second is the geometric restart strategy used in [5]. We start with a fixed restart interval of 100 conflicts and increase this interval by a constant factor of 1.5 after each restart. This means the solver restarts after 100 conflicts, then again after 150, then after 225, and so on.

## 6.6 Clause Deletion

The CDCL algorithm learns a new clause after every conflict. If all learned clauses were kept, their number could grow exponentially in the number of variables, which quickly becomes impractical. To address this, we follow a clause deletion strategy described in [11]. Our solver maintains a separate database for learned clauses and performs a garbage collection step every 2000 conflicts. Clauses of size at most 8 are always kept, while larger clauses are retained only if they contain at most one unassigned literal under the current assignment. We chose an interval of 2000 conflicts so that the solver incurs less overhead on most smaller puzzles. The threshold of 8 was selected because it proved effective in practice, while the condition of at most one unassigned literal is used because such clauses are unit and thus immediately useful for unit propagation.

# 7 Experimental Evaluation

The evaluation focuses on time-based performance across a wide range of Nonogram puzzles, comparing the two encodings, different configurations of our solver, and our solver against another solver.

## 7.1 Methodology

The evaluation is structured in three parts. First, we compare the two encodings in terms of their formula sizes, their encoding performance, and their solving performance. Second, we compare the different decision heuristics and restart strategies within our CDCL solver. Lastly, we compare the performance of our solver against MiniSat 1.14, whose source code is available at [6]. The choice of an older and less competitive solver is deliberate, as MiniSat 1.14 is based on the framework presented in [5], which also serves as the foundation for our own implementation.

The experiments are based on the dataset from the *Survey of Paint-by-Number Puzzle Solvers* [16], in which the performance of solvers of that time are compared on 2491 black-and-white puzzles. From this collection, we restrict ourselves to 1166 small- to medium-sized puzzles that have a unique solution, as these instances are representative of common Nonogram sizes and a large number of them is still computationally feasible for testing. The uniqueness of the solution is important because the Nonogram files provide only one reference solution, even if a puzzle has multiple solutions. Since we verify our results against these files, puzzles with more than one solution were excluded to ensure reliable correctness checking. The size distribution of the included puzzles is shown in Table 1.

| Puzzle size range | Number of puzzles |
|---|---|
| 0x0–10x10 | 110 |
| 10x10–20x20 | 582 |
| 20x20–30x30 | 474 |

Table 1: The size distribution based on the number of cells of the 2242 puzzles.

All experiments were carried out on a desktop machine running Windows 11, equipped with an AMD Ryzen 7 3700X processor and 16 GB of RAM. The solver and encodings were implemented in Java and executed on the Java Virtual Machine (JVM) with the default HotSpot JIT compilation. MiniSat 1.14 was written in C++ and was compiled and executed under the Windows Subsystem for Linux (WSL), with no additional optimization flags set.

## 7.2 Encodings Evaluation

We compare the Sequence Enumeration (SE) and Block-Based (BB) encodings on a restricted subset of our dataset containing only puzzles with at most 20 rows or 20 columns. This subset consists of 239 puzzles in total. We chose this restriction because the Sequence Enumeration encoding runs out of Java heap space on larger puzzles, and its encoding and solving times also become prohibitively long on such instances. Therefore, we opted to compare the encodings on this smaller subset.

The encodings are compared on the number of variables and clauses they produce, as shown in Table 2, and on the time required to generate the encoding and to solve the

encoded formula with our solver using the random decision heuristic and no restarts, as shown in Table 3. Encoding and solving were executed only once for each puzzle. For each puzzle, the solution returned by the solver was verified against the provided solution file. This verification step was not included in the time measurements, and in all cases the solutions were correct.

| | Variable count | | Clause count | |
|---|---|---|---|---|
| Size range | SE | BB | SE | BB |
| 1–9 | 3 | 3 | 3 | 3 |
| 10–99 | 20 | 26 | 4 | 8 |
| 100–999 | 92 | 164 | 28 | 80 |
| 1000–9999 | 119 | 46 | 91 | 148 |
| 10000–99999 | 5 | 0 | 111 | 0 |
| 100000–999999 | 0 | 0 | 2 | 0 |

Table 2: The distribution of variable and clause sizes for Sequence Enumeration (SE) and Block-Based (BB) encodings on the 239-puzzle subset.

| | Encoding time | | Solving time | |
|---|---|---|---|---|
| Time range (s) | SE | BB | SE | BB |
| 0.00000–0.00009 | 20 | 22 | 43 | 48 |
| 0.00010–0.00099 | 75 | 137 | 86 | 114 |
| 0.00100–0.00999 | 112 | 80 | 50 | 70 |
| 0.01000–0.09999 | 32 | 0 | 36 | 7 |
| 0.10000–0.19999 | 0 | 0 | 9 | 0 |
| 0.20000–0.49999 | 0 | 0 | 10 | 0 |
| 0.50000–0.99999 | 0 | 0 | 1 | 0 |
| 1.00000–3.99999 | 0 | 0 | 3 | 0 |
| 4.00000–9.99999 | 0 | 0 | 1 | 0 |

Table 3: The distribution of encoding and solving times for Sequence Enumeration (SE) and Block-Based (BB) encodings on the 239-puzzle subset. Solving was performed using the Random decision heuristic and no restarts.

The Block-Based encoding produces smaller formulas, has fewer variables, is faster to encode, and is faster to solve on the same set of puzzles, even for smaller instances. The higher encoding time of the Sequence Enumeration encoding can be explained by its larger number of variables, which correspond to the number line solutions that need to be generated for each row or column. Computationally, this is more demanding than generating the block start positions for the Block-Based encoding. The higher solving time of Sequence Enumeration is likewise explained by its larger number of clauses. Most of the solver's runtime is spent in unit propagation, and although we use watched literals, the propagation still needs to process more clauses than in the Block-Based encoding. Consequently, the smaller formulas of the Block-Based encoding result in both faster encoding and faster solving. Based on these results, we will use the Block-Based encoding for all subsequent experiments.

To complement these results, we also measured the encoding times for the Block-Based encoding on the entire dataset. Each puzzle was encoded only once. The distribution is shown in Table 4, and demonstrates that all puzzles can be encoded in under one second.

| Time range (s) | Block-Based encoding time |
|---|---|
| 0.00000–0.00009 | 30 |
| 0.00010–0.00099 | 171 |
| 0.00100–0.00999 | 886 |
| 0.01000–0.09999 | 79 |

Table 4: The distribution of encoding times in seconds for the Block-Based encoding on the full 1166 puzzles.

## 7.3 Decision Heuristics and Restart Strategies Evaluation

We compare the effect of decision heuristics and restart strategies for our solver on the full 1166 puzzles using the Block-Based encoding. We first solved all puzzles using our five decision heuristics without restarts. The heuristics are denoted as follows: Random (R), Line-Aware (L), Block-Length-Aware (B), VSIDS (V), and Sequential Order (O). After identifying Sequential Order as the best-performing heuristic, we combined it with Geometric restarts (OG) and Luby restarts (OL) and solved the puzzles again using these configurations. We define the best-performing heuristic as the one that solves the largest proportion of puzzles in under 0.01 seconds, following the approach of [16] and [2]. The full distribution of solving times is shown in Table 5, and the corresponding percentages of puzzles solved under 0.01 seconds are summarized in Table 6.

For all puzzles we verified that the solution returned by the solver matched the provided solution file. This verification step was not included in the measurements. Each puzzle was executed once with a timeout of 30 seconds. For all puzzles, the solutions produced by the solver matched the provided solutions.

| Time range (s) | R | L | B | V | O | OL | OG |
|---|---|---|---|---|---|---|---|
| 0.00000–0.00009 | 74 | 70 | 61 | 69 | 73 | 79 | 67 |
| 0.00010–0.00099 | 257 | 248 | 291 | 293 | 337 | 340 | 328 |
| 0.00100–0.00999 | 448 | 488 | 339 | 375 | 479 | 468 | 479 |
| 0.01000–0.09999 | 324 | 325 | 283 | 303 | 194 | 209 | 208 |
| 0.10000–0.19999 | 40 | 25 | 64 | 59 | 28 | 22 | 31 |
| 0.20000–0.49999 | 17 | 7 | 53 | 39 | 23 | 23 | 22 |
| 0.50000–0.99999 | 2 | 2 | 32 | 16 | 9 | 7 | 9 |
| 1.00000–3.99999 | 3 | 1 | 30 | 9 | 13 | 10 | 13 |
| 4.00000–9.99999 | 0 | 0 | 7 | 1 | 4 | 4 | 3 |
| 10.00000–29.99999 | 1 | 0 | 4 | 1 | 5 | 3 | 5 |
| $\geq$30 | 0 | 0 | 2 | 1 | 1 | 1 | 1 |

Table 5: Distribution of solving times in seconds for the Block-Based encoding on the full 1166 puzzles using different decision heuristics and restart strategies. The decision heuristics are Random (R), Line-Aware (L), Block-Length-Aware (B), VSIDS (V), and Sequential Order (O). The restart strategies are Line-Aware with Luby restarts (OL) and Line-Aware with Geometric restarts (OG).

| Heuristic | R | L | B | V | O | OL | OG |
|---|---|---|---|---|---|---|---|
| Solved <0.01s (%) | 66.8 | 69.1 | 59.3 | 63.2 | 76.2 | 76.0 | 74.5 |

Table 6: Percentage of the 1166 puzzles solved under 0.01 seconds for the Block-Based encoding using different configurations, derived from Table 5.

Based on our results, the heuristics can be ranked in the following order with respect to the proportion of puzzles solved under 0.01 seconds: Sequential Order performs best, followed by Line-Aware, Random, VSIDS, and Block-Length-Aware. When combined with restart strategies, Sequential Order with either Luby or Geometric restarts achieves almost identical performance to Sequential Order alone. The fact that the more frequent Luby restarts perform slightly better than the less frequent Geometric restarts suggests that the limited effect of restarts is not caused by the overhead of clearing the trail.

In terms of overall performance, only Line-Aware and Random succeed in solving all puzzles within the time limit. To further improve the performance of Line-Aware, we introduce a combined heuristic (C), which retains the Line-Aware strategy but, instead of selecting variables randomly when the line queue is empty, falls back to Sequential Order. We evaluated it under the same circumstances as the other heuristics, and the results can be found in Table 7. For all puzzles, the solutions produced by the solver matched the provided solutions.

| Time range (s) | C |
|---|---|
| 0.00000–0.00009 | 71 |
| 0.00010–0.00099 | 355 |
| 0.00100–0.00999 | 494 |
| 0.01000–0.09999 | 214 |
| 0.10000–0.19999 | 24 |
| 0.20000–0.49999 | 3 |
| 0.50000–0.99999 | 4 |
| 1.00000–3.99999 | 1 |
| 4.00000–9.99999 | 0 |
| 10.00000–29.99999 | 0 |
| ≥30 | 0 |

Table 7: Distribution of solving times in seconds for the Block-Based encoding on the full 1166 puzzles using the combined heuristic (C).

This combined heuristic achieves the best overall performance. It solves 78.9% of the puzzles in under 0.01 seconds, outperforming all other heuristics, and completes 99.9% of the puzzles in under one second. This is significantly better than both Random and VSIDS, which are general-purpose decision heuristics. Therefore, we conclude that considering structural properties of the Nonogram during solving can lead to an increase in performance compared to heuristics that do not exploit such structure.

Specifically for the Line-Aware and Sequential Order heuristics, the reason for the increased performance is likely that in the Block-Based encoding each variable corresponds to the placement of a block within a line. By prioritizing all block variables of a single line, the heuristic effectively fixes the coloring of that line before moving on. This directly restricts the possible block placements in all intersecting lines.

## 7.4 Evaluation against MiniSat

We compare our solver with MiniSat 1.14 on the full 1166 puzzles using the Block-Based encoding. As a reference for our solver, we use its fastest configuration from Section 7.3, namely the combined heuristic (C). For MiniSat 1.14, the encoding was written to DIMACS format, and solving times exclude parsing and file I/O. Each puzzle was run once with a timeout of 30 seconds. The results are shown in Table 8.

| Time range (s) | C | MiniSat 1.14 |
|---|---|---|
| 0.00000–0.00009 | 71 | 331 |
| 0.00010–0.00099 | 355 | 328 |
| 0.00100–0.00999 | 494 | 338 |
| 0.01000–0.09999 | 214 | 111 |
| 0.10000–0.19999 | 24 | 2 |
| 0.20000–0.49999 | 3 | 3 |
| 0.50000–0.99999 | 4 | 3 |
| 1.00000–3.99999 | 1 | 3 |
| 4.00000–9.99999 | 0 | 0 |
| 10.00000–29.99999 | 0 | 0 |
| $\geq$30 | 0 | 7 |

Table 8: Distribution of solving times in seconds for our solver with the combined heuristic (C) and MiniSat 1.14 on the full 1166 puzzles.

MiniSat performs better than our solver, solving 85.5% of the puzzles under 0.01 seconds. But it has 7 instances that are over the time limit. This is likely not due to its heuristic, since MiniSat employs VSIDS. As seen in Section 7.3, our solver with VSIDS has only 1 instance that goes over the time limit. Because our solver with VSIDS is also significantly slower than MiniSat, the likely reason for the performance difference is that our implementation is less optimized.

# 8 Conclusion

In this thesis we presented two novel SAT encodings for Nonograms. The Sequence Enumeration encoding represents each solution of a line by a variable, while the Block-Based encoding represents each block in a solution by a variable. To evaluate these encodings we built a CDCL solver from scratch. The goal was not only to compare the encodings, but also to investigate whether specific solver modifications can improve performance on Nonograms. For this purpose we integrated a binary fast path in the unit propagation, introduced four decision heuristics tailored to Nonograms, and tested two restart strategies with the aim of enhancing solving power.

The evaluation showed that the Block-Based encoding performs better than the Sequence Enumeration encoding in terms of formula size, encoding time, and solving time. It further demonstrated that decision heuristics exploiting structural properties of Nonograms can achieve clear performance improvements over general-purpose heuristics such as VSIDS. The best-performing heuristic is the combined Line-Aware and Sequential Order heuristic, which assigns all variables of a line before considering another variable for decision.

When compared with MiniSat, our solver is overall slower, which can be attributed to its less optimized implementation. Overall, the results show that both the choice of encoding and the design of heuristics have a strong influence on solver performance for Nonograms.

# References


[1] Ishan Akhouri. *Variable State Independent Decaying Sum (VSIDS) in SAT Solving*. Accessed: 2025-08-19. Mar. 2025. URL: https://medium.com/@theoreticalcs/building-a-smarter-sat-solver-integrating-vsids-7523473958fb.

[2] Abik Aramian and Varduhi Yeghiazaryan. "Solving Nonograms: A Constraint Satisfaction Approach". In: *Computers and Games. CG 2024. 12th International Conference, Virtual Event, November 25–29, 2024, Revised Selected Papers*. Ed. by Michael Hartisch, Chu-Hsuan Hsueh, and Jonathan Schaeffer. Vol. 15550. Lecture Notes in Computer Science. Springer, 2025. DOI: 10.1007/978-3-031-86585-5_11.

[3] Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. "Cache Conscious Data Structures for Boolean Satisfiability Solvers". In: *Journal on Satisfiability, Boolean Modeling and Computation* 6.1-3 (2009), pp. 99–120.

[4] Cayden R. Codel, Jeremy Avigad, and Marijn J. H. Heule. "Verified Encodings for SAT Solvers". In: *2023 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2023, pp. 141–151. DOI: 10.34727/2023/isbn.978-3-85448-060-0_22.

[5] Niklas Een and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing, SAT 2003*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2004, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37.

[6] Niklas Eén and Niklas Sörensson. *MiniSat*. Accessed: 2025-08-16. 2025. URL: http://minisat.se/MiniSat.html.

[7] Aaron S. Foote. *On the Complexity and Threshold Behavior of Playing Nonogram Puzzles*. Middletown, CT, Apr. 2024. DOI: 10.14418/wes01.1.2902. URL: https://digitalcollections.wesleyan.edu/islandora/complexity-and-threshold-behavior-playing-nonogram-puzzles.

[8] Jinbo Huang. "The effect of restarts on the efficiency of clause learning". In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence*. IJCAI'07. Hyderabad, India: Morgan Kaufmann, 2007, pp. 2318–2323.

[9] Tommi Junttila. *Overview – Propositional satisfiability and SAT solvers*. Online course notes, CS-E3220, Aalto University. Accessed: 2025-08-07. 2020. URL: https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/overview.html.

[10] Michael Luby, Alistair Sinclair, and David Zuckerman. "Optimal speedup of Las Vegas algorithms". In: *Information Processing Letters* 47.4 (1993), pp. 173–180. DOI: 10.1016/0020-0190(93)90029-9.

[11] Joao Marques-Silva, Inês Lynce, and Sharad Malik. "Conflict-Driven Clause Learning SAT Solvers". In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 131–153.

[12] Matthew W. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. IEEE, 2001, pp. 530–535.

[13] Knot Pipatsrisawat and Adnan Darwiche. *Rsat 1.03: SAT Solver Description*. Tech. rep. D–152. Automated Reasoning Group. Los Angeles, CA: Computer Science Department, UCLA, 2006.

[14] Lawrence Ryan. “Efficient Algorithms for Clause-Learning SAT Solvers”. MA thesis. Simon Fraser University, Feb. 2004.

[15] Richard P. Stanley. *Enumerative Combinatorics, Volume 1*. 2nd ed. Vol. 49. Cambridge Studies in Advanced Mathematics. Section 1.2. Cambridge: Cambridge University Press, 2012. ISBN: 978-1-107-01542-5.

[16] Jan Wolter. *Survey of Paint-by-Number Puzzle Solvers*. Accessed: 2025-08-16. 2025. URL: `https://webpbn.com/survey/`.