Bachelor's Thesis Computing Science

Pattern as an SAT problem

Nick van Oers
s1009378

May 17, 2024

First supervisor/assessor:
Dr. C.L.M. Kop

Second assessor:
Dr. J.S.L. Junges

Radboud University

Abstract

Pattern is a logic puzzle that has been proven to be NP-complete. In this thesis, we aim to encode the Pattern puzzle as a Boolean Satisfiability (SAT) problem. We show two ways of encoding the Pattern puzzle as an SAT problem and we use these encodings to develop two ways to generate Pattern puzzles. We will test the performance of both encodings and also of the generation methods. We find that the second encoding and the second generation method are the most efficient.

Contents

Chapter 1

Introduction

Logic puzzles have been popular pastimes for centuries, challenging players to exercise their problem-solving skills and logical reasoning. Pattern, also known as Nonogram or Japanese Puzzles, is a popular logic puzzle where players are tasked with filling in cells in a grid based on numerical clues provided for each row and column. The challenge lies in deducing which cells should be filled and which should be left blank, ultimately revealing a hidden picture.

Various logical puzzles, such as Sudoku and Pattern, are proven to be NP-complete. This means that there is no known efficient algorithm to solve these puzzles, and the time it takes to solve them grows exponentially with the size of the puzzle. However, solutions to these puzzles can be verified in polynomial time. To be NP-complete, a problem p must also be NP-hard, meaning that each problem in NP can be reduced to problem p in polynomial time.

In this thesis, we will use this notion of NP-completeness to reduce the Pattern puzzle to a SAT problem. SAT is famously known to be NP-complete, as it is the first problem that was proven to be NP-complete[7]. There are automatic solvers for SAT problems, which we will use to find solutions to Pattern puzzles.

We will first discuss some preliminary knowledge about Pattern and SAT problems. Then we will discuss a naive way to encode a Pattern puzzle as an SAT problem. Next, we will discuss an improved way of encoding a Pattern puzzle as an SAT problem. We will then look at two different ways how to generate Pattern puzzles. After that, we will discuss the experiments we have done on the various encodings and the generation algorithms. We will then discuss related works, summarize the research, and discuss future work.

Chapter 2

Preliminary knowledge

2.1 The Pattern puzzle

Pattern is a single-player logical puzzle where the player needs to find a hidden image with given clues. The game is played on a grid of squares, where each square is either black or white, combined with a set of clues. The player needs to fill in the grid with black and white squares, such that the clues are satisfied. The goal is to find the hidden image. Solving a Pattern puzzle is an NP-complete problem[11].

The player starts with an empty grid and is given a set of clues. Each clue corresponds to a row or column of the grid and contains one or more numbers. The individual numbers indicate a group of consecutive black squares in that row or column. In between each group of black squares, there must be at least one white square. All other squares in the row or column must be white.

The goal of the game is to satisfy all clues in the grid, such that the hidden image is revealed. A player can make a move by coloring a square in black or white. In Figure 2.1 you can see an example of a Pattern puzzle, with the starting grid on the left and the solution on the right.

2.2 Conjunctive Normal Form

Logical formulas are expressions that are composed of variables and operators that represent a logical relationship between the variables. Conjunctive Normal Form (CNF) is a way of representing logical formulas. A CNF formula consists of a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a variable or the negation of a variable. Example 2.2.1 is an example of a CNF formula. What is important for a formula to be in CNF, is that all clauses contain disjunctions and that each disjunction contains one or more literals. We can convert any logical formula to a CNF formula that is equisatisfiable. This conversion can be done in

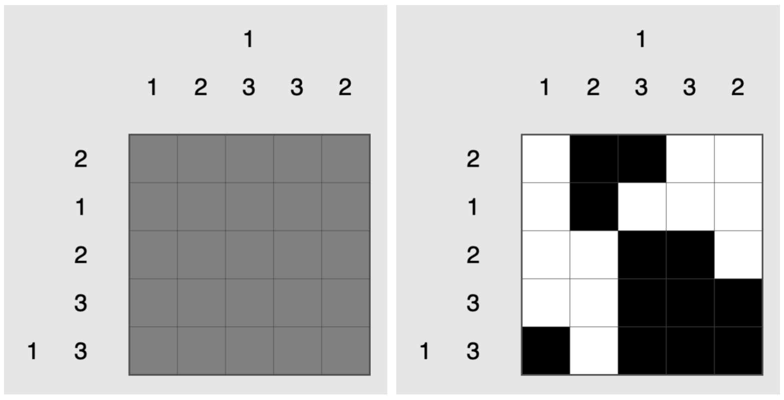


Figure 2.1: An empty Pattern puzzle and its solution

linear time using a Tseytin transformation(Section 2.4), which makes CNF formulas very useful for solving SAT problems.

Example 2.2.1. $F = (\neg A \vee \neg B) \wedge (B \vee C) \wedge D$

2.3 Boolean satisfiability problems and SAT solvers

A boolean satisfiability problem (hereafter called SAT) is a problem that determines whether given a logical formula, there exists a truth assignment that satisfies the formula. In a truth assignment of a CNF, each variable is assigned a specific value, and the entire formula is then evaluated to determine whether it is true or false. An example of a truth assignment is given in Example 2.3.1. SAT is proven to be NP-complete[12]. NP-complete is a complexity class that consists of problems that are at least as hard as any problem that is in NP.

Example 2.3.1. We take the formula $(\neg A \vee \neg B) \wedge (B \vee C) \wedge D$. A truth assignment where $A = False$, $B = False$, $C = True$ and $D = True$ satisfies the formula. A truth assignment where $A = False$, $B = False$, $C = True$, and $D = False$ does not satisfy the formula. However, the formula is satisfiable, because there exists a truth assignment that satisfies the formula.

An SAT solver is a tool that can determine whether a logical formula is satisfiable or not. It takes a CNF formula as input, and its output consists of whether there is a truth assignment that satisfies the formula. If such an assignment exists, it also outputs the found assignment. Whenever multiple assignments satisfy the formula, it outputs only one of them. This comes from the fact that an SAT solver is tasked with determining whether a formula is satisfiable, not with finding all assignments that satisfy a formula.

We will use SAT4J v2.3.6 as our SAT solver, which was first released in 2004. This SAT solver is an implementation of MiniSat[9]. This specific implementation accepts formulas in the DIMACS format[1]. Example 2.3.2 shows an example of a formula in DIMACS format. The first line contains the header `p cnf <variables> <clauses>`, where `<variables>` is the number of variables in the formula and `<clauses>` is the number of clauses in the formula. The following lines contain the clauses of the formula followed by a 0 signalling the end of the clause.

Example 2.3.2. Suppose we have the formula of Example 2.2.1, we can rewrite this to the following DIMACS format.

```
p cnf 4 3
-1 -2 0
2 3 0
4 0
```

2.4 Tseytin transformation

For an SAT solver to be able to solve a formula, it needs to be in CNF. However, it might occur that when creating a formula, you cannot go around the fact that it is not in CNF. For example, your formula might be in DNF, which is the disjunctive normal form[2]. Example 2.4.1 shows an example of a DNF formula. Converting a DNF formula to a CNF formula is very expensive computationally and leads to an exponential increase in the size of the formula. That is where we use the Tseytin transformation[15].

Example 2.4.1. $F = (A \wedge B) \vee (C \wedge D)$

The Tseytin transformation is a technique that can be used to convert any boolean formula to a CNF formula, in linear time relative to the input. Given a formula A, the Tseytin transformation converts A to a CNF formula B such that A and B are equisatisfiable. The key idea is that each subformula, except for literals, is given a name, which is then used as a fresh variable. Using those new variables and some rewrite rules, you can rewrite the subformulas to CNF. The resulting formula is a conjunction of these rewritten subformulas. The Tseytin transformation $T(A)$ of A is defined as the CNF of $x_n \leftrightarrow \neg a$ for every non-literal subformula of the shape $\neg a$, and as the CNF of $x_n \leftrightarrow (A_1 \diamond A_2)$ for every subformula of the shape $A_1 \diamond A_2$, for $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$. x_n is a fresh variable that is used to represent the subformula.

[1] `https://logic.pdmi.ras.ru/~basolver/dimacs.html`

[2] Similar to CNF, but a disjunction of conjunctions instead of the other way around

Formula	cnf()	CNF formula
$\neg y$	$cnf(x \leftrightarrow \neg y)$	$(x \vee y) \wedge (\neg x \vee \neg y)$
$y \wedge z$	$cnf(x \leftrightarrow (y \wedge z))$	$(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg x \vee z)$
$y \vee z$	$cnf(x \leftrightarrow (y \vee z))$	$(\neg x \vee y \vee z) \wedge (x \vee \neg y) \wedge (x \vee \neg z)$
$y \rightarrow z$	$cnf(x \leftrightarrow (y \rightarrow z))$	$(\neg x \vee \neg y \vee z) \wedge (x \vee y) \wedge (x \vee \neg z)$
$y \leftrightarrow z$	$cnf(x \leftrightarrow (y \leftrightarrow z))$	$(\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y \vee z)$

Table 2.1: Tseytin transformation rules

Example 2.4.2. We take the formula $F = (A \wedge \neg B) \vee C$. The first step is to find all the subformulas. We find $\neg B$, $A \wedge \neg B$, and $(A \wedge \neg B) \vee C$. We will represent them by variables x_1, x_2, and x_3 respectively. We then apply the transformation rules to each subformula.

$$\begin{array}{lr}
x_1 \leftrightarrow \neg B & \text{cnf}(x_1 \leftrightarrow \neg B) = (x_1 \vee B) \wedge (\neg x_1 \vee \neg B) \\
x_2 \leftrightarrow (A \wedge \neg B) & \text{cnf}(x_2 \leftrightarrow A \wedge x_1) = (x_2 \vee \neg A \vee \neg x_1) \wedge (\neg x_2 \vee A) \wedge (\neg x_2 \vee x_1) \\
x_3 \leftrightarrow ((A \wedge \neg B) \vee C) & \text{cnf}(x_3 \leftrightarrow x_2 \vee C) = (\neg x_3 \vee x_2 \vee C) \wedge (x_3 \vee \neg x_2) \wedge (x_3 \vee \neg C)
\end{array}$$

The final CNF formula consists of the conjunction of these CNF formulas.

$$\begin{aligned}
T(F) = & x_3 & \wedge \\
& (\neg x_3 \vee x_2 \vee C) \wedge (x_3 \vee \neg x_2) \wedge (x_3 \vee \neg C) & \wedge \\
& (x_2 \vee \neg A \vee \neg x_1) \wedge (\neg x_2 \vee A) \wedge (\neg x_2 \vee x_1) & \wedge \\
& (x_1 \vee B) \wedge (\neg x_1 \vee \neg B)
\end{aligned}$$

Chapter 3

A naive algorithm for Pattern puzzles

To solve a puzzle using an SAT solver, you would first need to convert the puzzle to a CNF formula. Doing so requires various steps and strategies to be taken. In this chapter, we will describe an initial naive solution to the problem of converting a Pattern puzzle to a CNF formula. This solution is not the final solution, but it is a good starting point to get a better understanding of the problem. From there we will find an improved solution that is more efficient in solving the problem.

3.1 Initialisation

To start the conversion to a CNF formula, we first need to initialize the basics of a puzzle. We need to know the size of the board in terms of rows and columns, and all of the constraints on each of the rows and columns.

3.2 Solving the puzzle

After the initialization of the puzzle, we can start solving it. We will solve the puzzle by using an SAT solver, but we can not just give the puzzle to an SAT solver as is. We need to find a way to convert the puzzle to a CNF formula, such that the SAT solver can process it.

All logic puzzles consist of a set of constraints and rules that need to be satisfied. Given these constraints, you can rewrite these to a boolean formula such that an SAT solver can solve it. Eventually, the puzzle comes down to evaluating each cell whether it should be black or white, considering the relevant constraints for that cell and the cells orthogonally adjacent to it. Each possible solution to each row or column can be converted to a logic formula by using a disjunction of all the possible configurations that a row or column can have. Creating a conjunction of all these disjunctions gives us a logical formula that represents the puzzle.

After we have converted the puzzle to a CNF formula, we can give it to an SAT solver. The SAT solver will then try to find a truth assignment for the formula, which solves the puzzle. To be able to give the formula to an SAT solver, we need to assign each cell a unique variable. Each cell gets a name $board_{row,column}$ that represents their position on the board in a single integer.

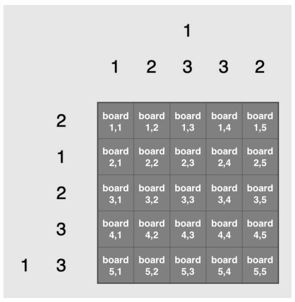


Figure 3.1: A board with a size of 5 by 5 with the names of the fields.

3.2.1 Generating possibilities

To generate the logic formulas for each row and column, we look at them individually. For each row or column, we look at its clue list and from there generate all different possibilities for that row or column. Let's look at the procedure for generating possibilities for a single row with size 15 and with a clue list of 6,3,2. We know from the clue list that there needs to be a total of 6+3+2=11 black cells in the row. That leaves us with 15-11=4 white cells. Our strategy to generate all the possibilities is to first represent each block of black cells as a single cell. We also know that each block of black cells needs to be separated by at least one white cell. Therefore we add a white cell after each block in the clue list, except for the last block since this can be at the end of the row or column. From there we are left with the remaining white cells that need to be placed in the row or column. These white cells need to be placed in between the blocks of black cells and we can assign an index to those places. To then obtain all possible placements of these white cells, we generate all increasing orderings of the indices with a maximum length of the number of white cells to place. For example, for a row with size 15 and with a constraint of 6,3,2, we first add a white cell after each block of black cells except for the last block. From there we are left with 2 white cells that can be placed in 4 places in the row. We generate all increasing orderings of the indices 1 through 4 with a length of 2 which we then use to place the white cells in the row. The possibilities for this row together with the orderings are shown in Example 3.2.1. We do this generation for each row and column and store these possibilities in a list.

Example 3.2.1. All possible increasing orderings of digits 1 through 4 with a length of 2 followed by the placing of the remaining white cells amongst the blocks.

{1,1}	`[0,0,1,1,1]`	{2,3}	`[1,0,1,0,1]`
{1,2}	`[0,1,0,1,1]`	{2,4}	`[1,0,1,1,0]`
{1,3}	`[0,1,1,0,1]`	{3,3}	`[1,1,0,0,1]`
{1,4}	`[0,1,1,1,0]`	{3,4}	`[1,1,0,1,0]`
{2,2}	`[1,0,0,1,1]`	{4,4}	`[1,1,1,0,0]`

3.2.2 Generating a CNF formula

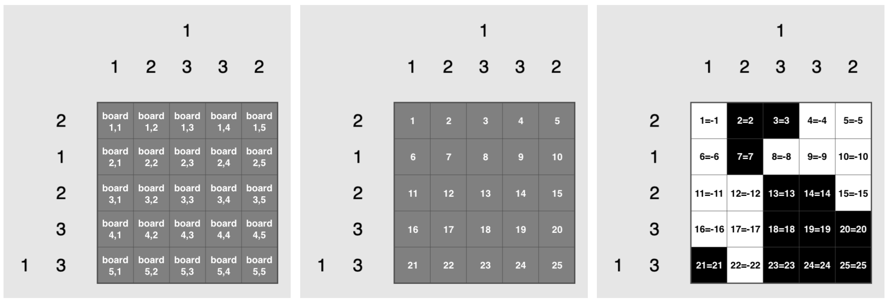
Now that we have all the possibilities for each row and column, we need to generate a CNF formula that represents the puzzle. The logic formula for a row or column is a disjunction of all the possibilities for that row or column. This is in the form of a DNF. Each one of those possibilities is then a conjunction of all the cells in that row or column. The conjunction of each logic formula for each row and column is the logic formula for a puzzle. However, this logic formula is not in CNF form, since you have a conjunction of DNFs. Therefore we need to convert this logic formula to CNF. We do this by using the Tseytin transformation. As mentioned in Section 2.4, the CNF formula that is a result of the Tseytin transformation, is equally satisfiable as the original formula.

The Tseytin transformation makes use of new free variables for each subformula. We want to use variables that make sense in the context of the puzzle. After applying the transformation, we will convert the formula to the DIMACS format which uses integer variables. We want the variables we use in the DIMACS format to represent the cells in the puzzle, and any extra variables we need to be assigned after that. We know that we have at most $r \cdot c$ cells in the puzzle, where r is the number of rows and c is the number of columns. Therefore we can use the integers from 1 to $r \cdot c$ for the cells in the puzzle and all integers from $r \cdot c + 1$ and up for the extra variables.

Now that we know which variables we can use, we can start applying the Tseytin transformation. Note that the outer conjunction does not need to be transformed, since it is already in CNF form. Therefore we only transform the inner disjunctions. We iterate over all of these and apply the Tseytin transformation to each of them as explained in Section 2.4. The result of this transformation is a CNF formula that is equivalent to the original logic formula.

3.2.3 Obtaining a solution

Now that we have a CNF formula we can almost give it to an SAT solver. First, we want to convert our formula into the DIMACS format. As said in section Section 3.2.2, we want to use the integers from 1 to $r \cdot c$ for the cells in the puzzle and all integers from $r \cdot c + 1$ and up for the extra variables. The extra variables already have the correct names, but we still need to convert the cell names to integers. We do this with the formula $(row - 1) \cdot c + column$, where row is the row of the cell, $column$ is the column of the cell, and c is the number of columns in the puzzle. We then put the formula in the DIMACS format and give it to the SAT solver. If the SAT solver returns a solution, we can convert the solution back to the puzzle format. We do this by iterating over all the cells in the puzzle and checking the cell's corresponding assigned variable in the solution. If the variable is set to true, we set the cell to black, otherwise, we set it to white. If the SAT solver returns that the formula is unsatisfiable, we know that the puzzle is unsolvable.



1
1 2 3 3 2
2 1 2 3 1 3
board 1,1 board 1,2 board 1,3 board 1,4 board 1,5
board 2,1 board 2,2 board 2,3 board 2,4 board 2,5
board 3,1 board 3,2 board 3,3 board 3,4 board 3,5
board 4,1 board 4,2 board 4,3 board 4,4 board 4,5
board 5,1 board 5,2 board 5,3 board 5,4 board 5,5
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
1=-1 2=2 3=3 4=-4 5=-5
6=-6 7=7 8=-8 9=-9 10=-10
11=-11 12=-12 13=13 14=14 15=-15
16=-16 17=-17 18=18 19=19 20=20
21=21 22=-22 23=23 24=24 25=25

Figure 3.2: A board with a size of 5 by 5 with the names of the fields converted to the DIMACS format and an illustration of the assignment of the variables.

Chapter 4

An improved algorithm for Pattern puzzles

We have already found a naive solution to the problem of solving a Pattern puzzle using an SAT solver. We can improve this solution to get a more efficient and faster algorithm. The problem with the naive algorithm is that it generates a very large CNF formula for larger puzzles. This slows down the algorithm by a significant amount. In this chapter, we will describe how the improved algorithm works and how it is implemented. We take the same approach of converting the puzzle to a CNF formula, however, we will make more use of the constraints and rules of the puzzle to get a more efficient formula.

4.1 Representation of a puzzle

To be able to convert the puzzle to a CNF formula we need to represent the puzzle in a way that we can easily convert it. We will use the same representation for fields as in the naive algorithm, however, we will add some extra information to make it easier to convert the puzzle. We will represent the puzzle as two lists of lines where each line is a list of fields. The first list of lines represents the rows of the puzzle and the second list of lines represents the columns of the puzzle. Each line has a number which we will use to refer to the line. The lines that represent the rows will be numbered from 1 to the number of rows in the puzzle. The lines that represent the columns will be numbered from the number of rows in the puzzle plus one to the number of rows plus the number of columns in the puzzle. If we have a puzzle of size 10x10, the rows will be numbered from 1 to 10 and the columns will be numbered from 11 to 20.

Besides the lines, we also need to represent the blocks. A block is a sequence of fields in a line that is connected to a clue of that line. We will represent the blocks as integers starting at 1 and ranging up to the number of blocks in the constraint. If we have a line with clue list `6 3 2`, the blocks will be numbered from 1 to 3.

4.2 Variables

Before we can start to convert the puzzle to a CNF formula, we need to define the variables that we will use. We will use three types of variables which are encoded in different ways.

1. The first type of variables are the variables that represent whether a specific field is the start of a certain block. We denote these variables as $is\_start\_of\_block(l, b, f)$ where l is the line, b is the block and f is the field.
2. The second type of variables are the variables that represent whether a specific field is part of a certain block. We denote these variables as $is\_part\_of\_block(l, b, f)$ where l is the line, b is the block and f is the field.
3. The third type of variables are the variables that represent whether a specific field is black. We denote these variables as $field\_is\_black(f)$ where f is the number of the field.

4.3 Functions

To be able to create a better encoding of the puzzle we need to formally define some aspects of the puzzle that we will be working with. We need to define several functions to help us with creating a readable encoding. We define the following functions:

- R: This function returns the list of rows in the puzzle.
- C: This function returns the list of columns in the puzzle.
- $L(l)$: This function returns the length of line l.
- $size(b, l)$: This function returns the size of block b in line l.
- $B(l)$: This function returns the list of blocks in line l.
- $F(l)$: This function returns the list of fields in line l.

4.4 Rules of the puzzle

Now that we have defined the predicates we can start to express the rules of the puzzle. All of the rules that we create are derived from the rules of the puzzle. We will start with the rules that are the most basic and then move on to the more complex rules.

1. If a field is the start of a block, it is also part of that block.

$$\bigwedge_{l=1}^{|R \cup C|} \bigwedge_{b=1}^{|B(l)|} \bigwedge_{f=1}^{|F(l)|} is\_start\_of\_block(l, b, F(l)[f]) \rightarrow is\_part\_of\_block(l, b, F(l)[f])$$

This implication is rather trivial, but it is important to include it in our encoding.

2. If a field is part of a block, it is black.

$$\bigwedge_{l=1}^{|R \cup C|} \bigwedge_{b=1}^{|B(l)|} \bigwedge_{f=1}^{|F(l)|} is\_part\_of\_block(l, b, F(l)[f]) \rightarrow field\_is\_black(F(l)[f])$$

When combining these two rules you can see the importance of the first one. We know that if a field is part of a block, or if it is the start of a block, it is black. But we also know that if a field is the start of a block, it is also part of that block. Using the above two rules we can deduce all three of these statements.

3. Each block starts at a field such that the block fits in the row.

$$\bigwedge_{l=1}^{|R \cup C|} \bigwedge_{b=1}^{|B(l)|} \bigvee_{f=1}^{|F(l)|-size(b)+1} is\_start\_of\_block(l, b, F(l)[f])$$
$$\bigwedge_{l=1}^{|R \cup C|} \bigwedge_{b=1}^{|B(l)|} \bigwedge_{f=L(l)-size(b)+2}^{|F(l)|} \neg is\_start\_of\_block(l, b, F(l)[f])$$

The first part of this rule denotes the fields where a block can start such that it fits in the line. The second part of the rule denotes the fields where a block cannot start such that it fits in the line. For example, if we have a block of size 3 and we are looking at a line of size 5, the first part of the rule tells us that the block can start at fields 1, 2, and 3. The second part tells us that the block cannot start at fields 4 and 5.

4. If a block starts at a field, the next fields are also part of that block.

$$\bigwedge_{l=1}^{|R\cup C|}\bigwedge_{b=1}^{|B(l)|}\bigwedge_{f=1}^{|F(l)|}\bigwedge_{f'=f+1}^{f+size(b)} is\_start\_of\_block(l,b,F(l)[f]) \rightarrow \\ is\_part\_of\_block(l,b,F(l)[f'])$$

This rule states that when we have a block of size b and we know that it starts at field f, we can deduce that the next $b-1$ fields are also part of that block.

5. If a block starts at a field, it does not start at another field.

$$\bigwedge_{l=1}^{|R\cup C|}\bigwedge_{b=1}^{|B(l)|}\bigwedge_{f=1}^{|F(l)|}\bigwedge_{f'=1}^{|F(l)|} is\_start\_of\_block(l,b,F(l)[f]) \rightarrow \\ \neg is\_start\_of\_block(l,b,F(l)[f']) \text{ for } f\neq f'$$

6. If a block starts at a field, if it exists, the field before it is white.

$$\bigwedge_{l=1}^{|R\cup C|}\bigwedge_{b=1}^{|B(l)|}\bigwedge_{f=2}^{|F(l)|} is\_start\_of\_block(l,b,F(l)[f]) \rightarrow \\ \neg field\_is\_black(F(l)[f-1])$$

7. The first field after a block is white if it exists.

$$\bigwedge_{l=1}^{|R\cup C|}\bigwedge_{b=1}^{|B(l)|}\bigwedge_{f=1}^{|F(l)|-size(b)} is\_start\_of\_block(l,b,F(l)[f]) \rightarrow \\ \neg field\_is\_black(F(l)[f+size(b)])$$

Rules 6 and 7 state that the fields surrounding a block, if they exist, should be white. This is one of the basic rules of the puzzle.

8. If a block starts at field f, the fields before that are not part of that block.

$$\bigwedge_{l=1}^{|R\cup C|}\bigwedge_{b=1}^{|B(l)|}\bigwedge_{f=1}^{|F(l)|}\bigwedge_{f'=1}^{f-1} is\_start\_of\_block(l,b,F(l)[f]) \rightarrow \\ \neg is\_part\_of\_block(l,b,F(l)[f'])$$

9. If a block ends at field $f + size(b)$, the fields after that are not part of that block

$$\bigwedge_{l=1}^{|R \cup C|} \bigwedge_{b=1}^{|B(l)|} \bigwedge_{f=1}^{|F(l)|} \bigwedge_{f'=f+size(b)}^{|F(l)|} is\_start\_of\_block(l, b, F(l)[f]) \rightarrow \neg is\_part\_of\_block(l, b, F(l)[f'])$$

We know that if a block starts at a field f, only the following $size(b) - 1$ fields are part of that block. From there we can deduce that the fields before f and after $f + size(b) - 1$ are not part of that block.

10. If a field is black, it is part of some block in its row and column.

$$\bigwedge_{r=1}^{|R|} \bigwedge_{c=|R|+1}^{|C|} field\_is\_black(F(c)[r]) \rightarrow \bigvee_{b=1}^{|B(r)|} is\_part\_of\_block(r, b, F(c)[r])$$
$$\bigwedge_{r=1}^{|R|} \bigwedge_{c=|R|+1}^{|C|} field\_is\_black(F(c)[r]) \rightarrow \bigvee_{b=1}^{|B(c)|} is\_part\_of\_block(c, b, F(c)[r])$$

We already stated that if a field is part of a block, it is black. However, we also know that if a field is black, it is part of a block. This rule states that if a field is black, it is part of a block in its row and column.

You also see that we get the number of the field using $F(c)[r]$ instead of $F(l)[f]$ which we have been using so far. This is because we currently do not have dedicated l and f values. However, we can use our r and c values to get the number of the field, using the way the numbering of the rows and columns is set up. The numbering of the rows starts at 1 and goes up to $|R|$, this corresponds with the list of indices of the columns. We can use this to get the number of the r^{th} field in line c, $F(c)[r]$.

11. If a block starts at a field, the block after that cannot start before or at that field.

$$\bigwedge_{l=1}^{|R \cup C|} \bigwedge_{b=1}^{|B(l)|} \bigwedge_{f=1}^{|F(l)|} \bigwedge_{f'=1}^{f} is\_start\_of\_block(l, b, F(l)[f]) \rightarrow \neg is\_start\_of\_block(l, b+1, F(l)[f'])$$

The way the blocks are ordered in the constraint is also in what order they should be placed in the puzzle. This rule enforces that if a block starts at a field, the block after that cannot start before or at that field. Since this rule is applied to all blocks, it also enforces that the blocks are placed in the correct order.

4.5 Generating a CNF formula

To be able to solve the puzzle, we need to generate a CNF formula that we can give to an SAT solver. We will use the rules that we have defined in the previous section to generate the CNF formula. However, we cannot just make one large conjunction of all the rules since most rules contain implications and those cannot be in a CNF formula.

A	B	A→B	¬A∨B
T	T	T	T
T	F	F	F
F	T	T	T
F	F	T	T

Table 4.1: Truth table for an implication and an equivalent disjunction

Table 4.1 shows an equivalent formula for an implication. We see that it is a disjunction that is convenient to use in a CNF formula, and there is no need to introduce new variables. Using this solves our problem of having implications in our CNF formula.

As we can see in our rules, there are mostly conjunctions of implications. If we convert those implications to disjunctions, we obtain conjunctions of disjunctions which is already a CNF. In rule 3 we do not have any implications, but that rule is already written as a CNF formula. The only thing that we need to do to convert these rules into a proper CNF formula, is to make one large conjunction of all rules.

As opposed to our naive encoding in Section 3.2.2, we do not have to use the Tseytin Transformation to convert our rules to CNF.

4.6 Conversion to DIMACS

In this algorithm, we will convert our CNF formula with our encoded variables into a DIMACS format. We know that the DIMACS format uses a list of consecutive variables, which is not the case for our encoding. Therefore we need to convert our variables to a list of consecutive variables, which is done in various places of the algorithm for efficiency. We maintain a list of used variables. When we want to use a variable, we check if it is already in the list. If it is not, we append it to the list.

Before we can create the list, we need to name our variables. For the first two types of variables, $is\_start\_of\_block(l, b, f)$ and $is\_part\_of\_block(l, b, f)$, we will simply use the same name as its type and replace l, b, and f with their corresponding values. For variables of type $field\_is\_black(f)$ we will use only the field number as its name.

DIMACS files only accept consecutive digits higher than 0 as variable names. Whenever we want to use a variable in the DIMACS file, we take the index of that variable in the list of used variables and add 1 to it to get the name of the variable used in the DIMACS file.

Algorithm 1 shows how we convert the CNF formula to a DIMACS file.

Algorithm 1 Convert CNF formula to DIMACS

```
Require: $CnfFormula, variableList, DimacsFile$
  for $CnfClause$ in $CnfFormula$ do
    $DimacsClause \leftarrow []$
    for $lit$ in $CnfClause$ do
      if $lit$ not in $variableList$ then
        $variableList.append(lit)$
      end if
      $DimacsClause.append(variableList.index(lit) + 1)$
    end for
    $DimacsFile.append(DimacsClause)$
  end for
```

We can now give this DIMACS file to the SAT solver which will tell us whether there exists a solution for the generated CNF formula and thus the puzzle.

Chapter 5

Generation of Pattern puzzles

In this chapter, we will discuss the generation of Pattern puzzles. We will discuss two strategies for generating Pattern puzzles. The first strategy is a simple approach, which is efficient but generates arbitrary puzzles that you would not encounter as quickly when solving pre-made puzzles. The second strategy is an improved approach, which is slightly more complex but results in puzzles that are more comparable to pre-made puzzles.

5.1 Generation strategies

5.1.1 First strategy: full random generation

The first strategy for generating Pattern puzzles is to generate a random puzzle. This is done by creating a grid of a given size and then filling in each field in the grid randomly. When you have a solved puzzle, you can determine the approximate percentage of black fields in the puzzle. We convert the percentages to values between 0 and 1, rounding to one decimal place, and use this as a threshold. We call this threshold the density of the puzzle. To generate a random puzzle, we fill in each field in the grid with a black field with a probability equal to a given density. This is done by generating a random number between 0 and 1 for each field, and if the number is less than the density, we fill in the field as black, otherwise, we fill in the field as white.

When we have obtained a filled grid, we need to determine the clues that are needed to solve this puzzle. We iteratively go through each row and column of the grid, and for each row and column, we determine the lengths of the sequences of black fields. These values are stored in lists which form the clues for the puzzle.

This approach is very simple and efficient, but it has some drawbacks.

The main drawback is that the puzzles generated by this approach are not very representative of the puzzles that are found in the wild. Puzzles can not be very cohesive. With this, we mean that the black fields are not very connected. Another drawback is that when using lower densities, puzzles have a high chance of being ambiguous, meaning that there are multiple solutions to the puzzle, which can leave a human solver confused about how to solve the puzzle. On the other side, when using higher densities, puzzles have a high chance of being trivial, meaning that the solution can be determined without any effort. One of the advantages of this approach is that each puzzle is guaranteed to have a solution, as the actual puzzle is generated from a filled-in grid.

5.1.2 Second strategy: clue generation

The second strategy for generating Pattern puzzles is to generate a puzzle by first generating the clues, and then filling in the grid based on these clues. With this approach, we strive to achieve puzzles that are more representative of the puzzles that are found in the wild. For each row, we will determine the amount of clues, and then determine the size of each clue. Then we will generate the clues for the columns accordingly.

Before we can generate the clues, we first want to get some more information about the puzzles that are found in the wild. We have collected a dataset of 5000 puzzles, and we have analyzed these puzzles to get some statistics about the puzzles. For each size of the puzzle, we analyzed the average density, the frequency of the different lists of clues, and the frequency of individual clues. Table 5.1 is a summary of the statistics that we have found.

We will use these statistics to generate the clues for the puzzles. For any puzzle with a size for which we have the statistics, we can generate the clues per row of the puzzle. First, we need to process the data from Table 5.1 to make it usable for generating the clues. We will store the data in a separate dictionary for each size. The dictionary will contain two values, one for the clue list length and one for the clue size, which in turn contains another dictionary with the values from Table 5.1. However, we alter the data such that key $x + 1$ has the value of key x plus the value of key $x + 1$. We will use this later to randomly generate the clues for the puzzle. If we take the data from Table 5.1 for size 5, we get the following dictionary:

```
{
    "clueLength": {
        1: 7760,
        2: 9855,
        3: 10000,
    },
    "clueSize": {
        1: 4544,
        2: 8531,
        3: 11429,
        4: 12385
    }
}
```

In `clueLength`, the keys represent the length of the list of clues, and the values represent the number of times a list of clues of that length is found in the dataset. In `clueSize`, the keys represent the size of the individual clues, and the values represent the number of times an individual clue of that size is found in the dataset.

Now that we have processed the data, we can generate the clues for the puzzle. Algorithm 2 shows the algorithm to determine the clues for the rows of a puzzle, *clueDict* is the dictionary that contains the statistics for the puzzles found in the wild, and *puzzleSize* is the size of the puzzle we want to generate. We will generate the clues for each row of the puzzle, and then we will generate the clues for each column of the puzzle based on the clues for the rows.

To generate the clues for a row, we start by determining the length of the list of clues that we want to generate. We do this by generating a random number between 1 and the highest value in the clueLength key of the dictionary. We then take the clueLength key of the dictionary. From that inner dictionary, we return the key for the smallest value that is greater than or equal to the random number. That key is the length of the list of clues for the row. We then generate the individual clues for the row. This goes similarly to the generation of the length of the list of clues, but with the additional constraint that the total size of the clues can not exceed the size of the row. We repeat this process for each row of the puzzle.

Algorithm 2 Determine the clue list for a row

```
Require: $clueDict$, $puzzleSize$
  procedure GENERATECLUELENGTH
      $clueCount \leftarrow clueDict[''clueLength''].maxValue()$
      $clueLengthIndex \leftarrow getRandomNumber(1, clueCount)$
      for $key$ in $clueDict[''clueLength'']$ do
          if $clueLengthIndex \leq clueDict[''clueLength''][key]$ then
              return $key$
          end if
      end for
  end procedure
  procedure GETCLUESIZE($maxClueSize$)
      $clue \leftarrow 0$
      while $clue$ is 0 do
          $clueCount \leftarrow clueDict[''clueSize''].maxValue()$
          $clueSizeIndex \leftarrow getRandomNumber(1, clueCount)$
          for $key$ in $clueDict[''clueSize'']$ do
              if $clueSizeIndex \leq clueDict[''clueSize''][key]$
              and $key \leq maxClueSize$ then
                  $clue \leftarrow key$
              else
                  break
              end if
          end for
      end while
      return $clue$
  end procedure
  $clueList \leftarrow []$
  $clueLength \leftarrow generateClueLength$
  $maxBlackFields \leftarrow puzzleSize - (clueLength - 1)$
  for $i$ in range $(0, clueLength)$ do
      $maxClueSize \leftarrow maxBlackFields - (clueLength - i - 1)$
      $clue \leftarrow getClueSize(maxClueSize)$
      $maxBlackFields \leftarrow maxBlackFields - clue$
      $clueList.append(clue)$
  end for
```

Size		5	10	15	20	25	30	35	40
Average density		0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Clue list length	1	7760	7765	4198	1709	672	227	88	29
	2	2095	9814	13619	10658	6208	3084	7593	566
	3	145	2165	9419	16156	16242	12185	7593	3936
	4		240	2368	8612	16303	19699	17788	13139
	5		15	362	2415	7780	15478	21010	21825
	6			32	385	2273	6870	14118	20836
	7			2	60	427	1947	5824	12444
	8				5	86	430	1720	5138
	9					9	67	424	1621
	10						12	58	377
	11						1	5	76
	12								12
	13								1
Clue size	1	4544	10099	19917	33183	49842	69349	92654	119610
	2	3987	7287	13047	20820	30483	41344	54056	68236
	3	2898	7291	14407	24288	36576	51580	68307	88969
	4	956	4128	8324	14114	21151	30021	40118	51901
	5		2642	5539	9233	14146	20100	26527	34763
	6		1687	3781	6642	10260	14408	19609	25405
	7		994	2365	4361	6757	9850	13656	17747
	8		548	1453	2844	4687	6840	9303	12241
	9		252	1003	1870	3219	4739	6498	8414
	10			573	1261	2038	3208	4630	5924
	11			362	803	1435	2133	3173	4330
	12			211	530	928	1527	2182	2852
	13			135	327	642	1019	1483	2086
	14			64	206	424	656	1014	1467
	15				124	272	436	742	977
	16				80	178	311	515	727
	17				50	89	218	313	487
	18				27	66	142	243	349
	19				13	38	87	150	262
	20					35	72	94	165
	21					25	40	72	93
	22					15	27	59	72
	23					11	24	35	49
	24					5	10	16	24
	25						6	20	28
	26						3	7	15
	27						4	9	12
	28						4	4	9
	29						1	2	5
	30							3	4
	31							2	1
	32							2	2
	33							1	2
	34							1	1
	39								1

Table 5.1: Summary of the statistics of the puzzles found in the wild.

Before we can generate the clues for the columns of the puzzle, we need to determine where the blocks corresponding to the clues are placed. Figure 5.1 shows how we determine the place of the black fields corresponding to the clues in a row. We first look at the available fields for the first clue. We take the full length of the row, subtract the total size of the remaining clues, and subtract one more for each remaining clue. In Figure 5.1 we subtract 2 for the total size of the remaining clues, and 2 more for the number of remaining clues. We then randomly place the black fields in the row, fill the preceding fields and the first following fields with white fields, and then move on to the next clue. We then repeat the process of determining the available fields, randomly placing the black fields and placing the white fields for the remaining clues.

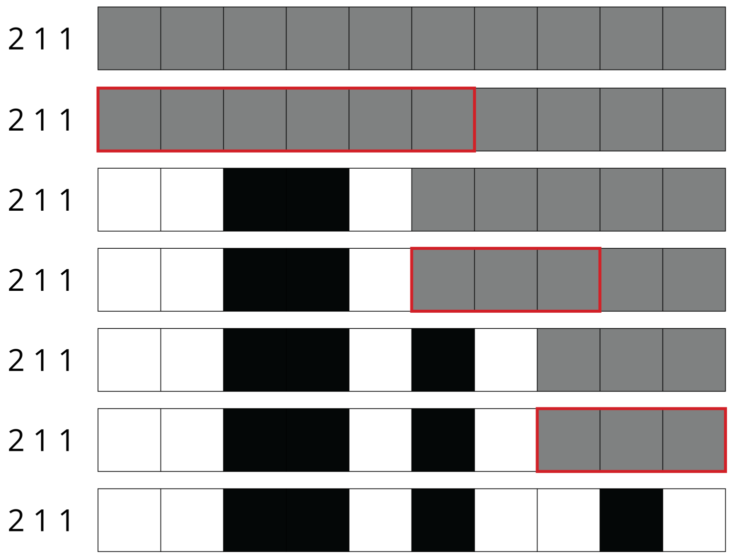


Figure 5.1: The process of determining the placement of each block in a row.

Now that we have determined where the black fields are in each row of the puzzle, we can determine the clues for the columns of the puzzle. We do this in the same way as in Section 5.1.1. We iteratively go through each row and column of the grid, and for each column, we determine the lengths of the sequences of black fields. These values are stored in lists which form the clues for the puzzle. Because we know where each clue sits in their row, we also know which fields are black and which are white. With this information, we can fill in the grid and determine the clues for each column.

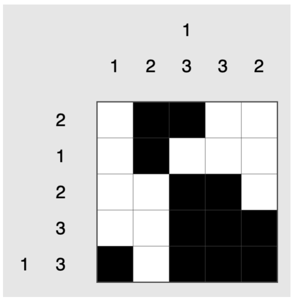
This approach is slightly more complex than the first approach, but it results in puzzles that are more representative of the puzzles that are found in the wild. There is no clear argument that makes the solution of the puzzles more cohesive since the placement of the black fields is still random, but the formation of the clues is more representative of the puzzles that are found in the wild. The main advantage of this approach is that the puzzles generally have a similar density to puzzles found in the wild. Because the number of black fields is determined by the clues, and the generation of clues in this approach is based on the statistics of the puzzles found in the wild, the puzzles generated by this approach are more comparable to the puzzles found in the wild. However, this approach also has the drawback that the puzzles generated are not guaranteed to be unambiguous, meaning that there is a chance that there are multiple solutions to a puzzle.

5.2 Uniqueness of puzzles

When generating puzzles, we want to make sure that the puzzles are unique, meaning that there is only one solution to the puzzle. This makes the puzzle solvable, as the solver can be sure that the solution is the correct one. When generating puzzles using either of our strategies, we can not guarantee that the puzzles are unique. Therefore we need to implement some measure that determines whether a puzzle has a unique solution or not. For this, we need to determine whether a puzzle has multiple solutions or not. We start by taking any solution to the puzzle; we name this solution A. There are multiple solutions to the puzzle if we can find a solution B to the puzzle that satisfies all of the clues but is not equal to A. We define a solution to a puzzle to be as described in Section 3.2.3

We can find another solution using an SAT solver. We already have a solution A for a generated puzzle, and we can use this solution to try to find a second solution B. We find solution B using an SAT solver and the algorithm described in Chapter 4. Before we give the DIMACS file to the SAT solver, we add a clause that states that the solution must be different from A. In solution B, any fields can be different so we add one clause that states that one of the fields must be different from the solution in A. Example 5.2.1 shows a solved puzzle and what clause we add to the CNF of a puzzle to find solution B.

Example 5.2.1. On the left is a solved puzzle, and on the right is a clause that states that a second solution must be different from the known solution.



$$\begin{array}{rrrrr}
1\lor & \neg 2\lor & \neg 3\lor & 4\lor & 5\lor \\
6\lor & \neg 7\lor & 8\lor & 9\lor & 10\lor \\
11\lor & 12\lor & \neg 13\lor & \neg 14\lor & 15\lor \\
16\lor & 17\lor & \neg 18\lor & \neg 19\lor & \neg 20\lor \\
\neg 21\lor & 22\lor & \neg 23\lor & \neg 24\lor & \neg 25
\end{array}$$

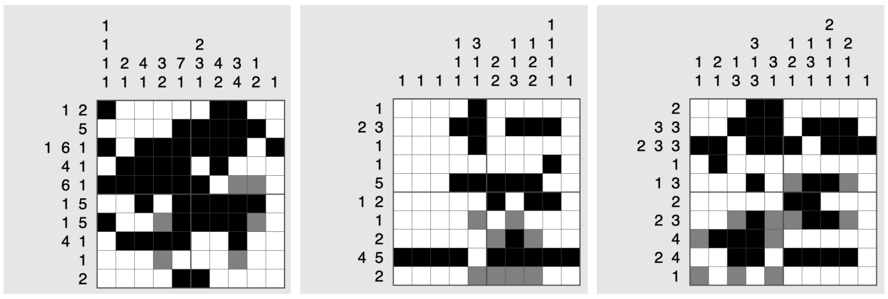


Figure 5.2: Three puzzles with different placements of variable fields.

We then use an SAT solver to find a solution to the puzzle with the added clause. If the SAT solver returns a solution, we know that the puzzle has multiple solutions. If the SAT solver does not return a solution, we know that the puzzle has a unique solution.

5.3 Making puzzles unique

When generating puzzles, we want to make sure that the puzzles are unique, meaning that there is only one solution to the puzzle. When generating puzzles using either of our strategies, we can not guarantee that the puzzles are unique. Therefore we want to develop some algorithm that changes puzzles that are not unique into unique puzzles.

5.3.1 Determining variable fields

We know that when a puzzle has multiple solutions, multiple fields can be either black or white. The goal of the algorithm is to find these variable fields and change the accompanying clues in such a way that the puzzle has a unique solution. We can find these variable fields by using the solutions we obtained from the SAT solver. We take the first solution we find, and we use that as our baseline. For each new solution we find, we compare it to the baseline solution. Any field that differs from the baseline solution is variable and is stored in a list. Eventually, we end up with a list of all variable fields in the puzzle.

5.3.2 Changing the clues

Now that we know which fields are causing the puzzle to have multiple solutions, we can change the clues in such a way that the puzzle has a unique solution. To determine how the clues need to be changed, we take a look at where the variable fields are located in the puzzle. More specifically, we look at how the variable fields are placed in comparison to each other.

Figure 5.2 shows three puzzles with their variable fields greyed out. You can see three different ways in which the variable fields are placed in the puzzle. In the first puzzle, the number of variable fields in each row and column is equal. In the second puzzle, the number of variable fields in each column is equal, but the number of variable fields in each row is not equal. In the third puzzle, the number of variable fields differs in each row and column. We can use this information to determine how the clues need to be changed. For each of the three different placements of variable fields, we have a different way of changing the clues.

1. If the number of variable fields for each row and column is equal, we fill the variable fields in one of the rows or columns with the highest number of variable fields with black fields and the rest of the variable fields with white fields.

2. If the variable fields are placed in such a way that each row has the same amount of variable fields, but the columns do not, we take the column with the most variable fields and fill those with black fields and the rest with white fields. We can do the opposite if the columns have the same amount of variable fields, but the rows do not.

3. If the variable fields are placed in such a way that neither the rows nor the columns have the same amount of variable fields, we take the row or column with the most variable fields and fill those with black fields and the rest with white fields.

Besides the placement of the variable fields, we also need to take into account the other fields in each row and column. It could happen that when we change the clues, we create a row or column that has no black fields. According to the rules of the puzzle, there must be at least one black field in each row and column. To avoid this issue, we need to make sure that each row and column has at least one black field apart from the variable fields. If this is the case, we can use one of the three ways to change the clues. If this is not the case, we use the empty row to move the clues, regardless of the placement of the variable fields. In case we have multiple empty rows or columns, we fill all of them.

Using these strategies to change the clues, we decrease the number of puzzles generated by our algorithm from Section 5.1.2 that have multiple solutions. In Section 6.2 you can see the results of the experiments run with these strategies.

The reasoning behind this strategy is that we wanted to change the puzzle as little as possible while trying to keep the amount of black fields in the puzzle as close to the original. We started with the simple example of four variable fields aligned in a square where there were two solutions possible; one with the top left and bottom right fields black and the other

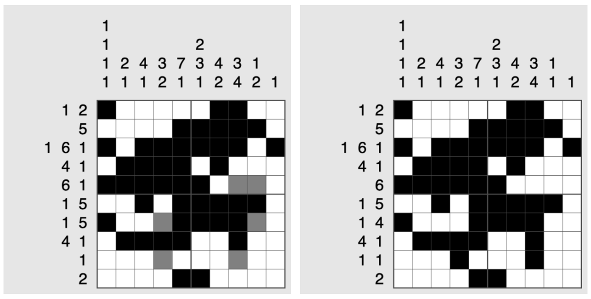


Figure 5.3: On the left is a puzzle with multiple solutions, and on the right is the left puzzle but with a few clues altered such that it is uniquely solvable.

with the top right and bottom left fields black. If we take both solutions and look at them from a column perspective, we see that both columns have at least one black field in each column. We can move one black field from the top row to the bottom row, and that perspective would be the same. However, looking at the full puzzle we see that the puzzle is now uniquely solvable after moving one black field to the bottom row. We expanded this idea to the full puzzle and various configurations of variable fields to come up with a strategy that makes unique puzzles out of most non-uniquely solvable puzzles.

Chapter 6

Experiments

In this chapter, we will look at the experiments that were conducted to evaluate the performance of the two algorithms proposed to solve puzzles, as well as the experiments conducted to evaluate the performance of the generation algorithms. As explained before, we have two different algorithms to solve puzzles, one naive algorithm and one improved algorithm. Because of the naive nature of the first algorithm, it has significantly more clauses, and thus worse performance, than the improved algorithm.

We will look at solving puzzles using both algorithms and compare the performance of the two algorithms. They will be evaluated in terms of the number of clauses and the time it takes to solve the puzzles, compared to their size. We will also look at the performance of the generation algorithms. They will be evaluated in terms of the time it takes to generate a puzzle and whether the generated puzzle is uniquely solvable, compared to its size. Lastly, we will look at making puzzles that have multiple solutions into puzzles that have a unique solution, and evaluate the performance of our algorithm in terms of time and success rate.

6.1 Comparing both algorithms

To compare both algorithms we need a set of puzzles that are guaranteed to have a unique solution. We will use Simon Tatham's Portable Puzzle Collection [4] to generate puzzles of different sizes. We have downloaded 1000 puzzles each of size $n * n$ for $n \in \{5, 10, 15, 20, 25, 30, 35, 40\}$. These puzzles are made to be humanly solvable so we expect our algorithms to be able to solve them in a reasonable amount of time. We let the algorithms solve the puzzles and we noted the time it took them to generate the full CNF formula and how long it took the SAT solver to find a solution to the formula. When first testing out the process of this experiment, we noticed that the naive algorithm took a long time to solve the puzzles, especially of sizes 25 and above. Therefore we decided to adhere to a time limit for each size and not consider puzzles of sizes 30 and higher for the naive algorithm. The improved algorithm is significantly faster thus we did not need to adhere to a time limit for it to get reasonable results.

Table 6.1 shows the size of the puzzle, which algorithm was used, the time limit, the average time it took to generate the CNF formula, the average time it took the SAT solver to solve the formula, and the percentage of puzzles that were solved within the time limit. Important to note, is that all puzzles both algorithms attempted to solve were solved successfully.

Figure 6.1 shows a graph of the total time it took to solve the puzzles of different sizes using both algorithms.

We can see that overall the improved algorithm is faster than the naive algorithm. Only for smaller sizes, the improved algorithm is slower than the naive algorithm; this is because the improved algorithm has to generate a larger CNF formula than the naive algorithm for smaller puzzles.

Size	Algorithm	Limit (ms)	Generation (ms)	Solving (ms)	Success Rate
5x5	Naive	500	2.83	1.02	100%
5x5	Improved	-	6.08	0.05	100%
10x10	Naive	500	21.92	1.06	100%
10x10	Improved	-	26.63	0.14	100%
15x15	Naive	500	223.32	7.61	99.6%
15x15	Improved	-	90.37	0.98	100%
20x20	Naive	4000	2973.33	124.51	85.9%
20x20	Improved	-	249.5	6.24	100%
25x25	Naive	50000	38105.71	1942.56	51%
25x25	Improved	-	770.78	34.87	100%
30x30	Naive	-	-	-	-
30x30	Improved	-	1515.35	129.89	100%
35x35	Naive	-	-	-	-
35x35	Improved	-	2562.76	351.56	100%
40x40	Naive	-	-	-	-
40x40	Improved	-	4283.21	937.39	100%

Table 6.1: Performance results of both algorithms on puzzles of different sizes.

6.2 Generation of puzzles

To evaluate the performance of the generation algorithms, we will look at the time it takes to generate a puzzle and whether the generated puzzle is uniquely solvable. We let each generation algorithm generate 500 puzzles of different sizes, $n * n$ for $n \in \{5, 10, 15, 20, 25\}$. The algorithms first generate

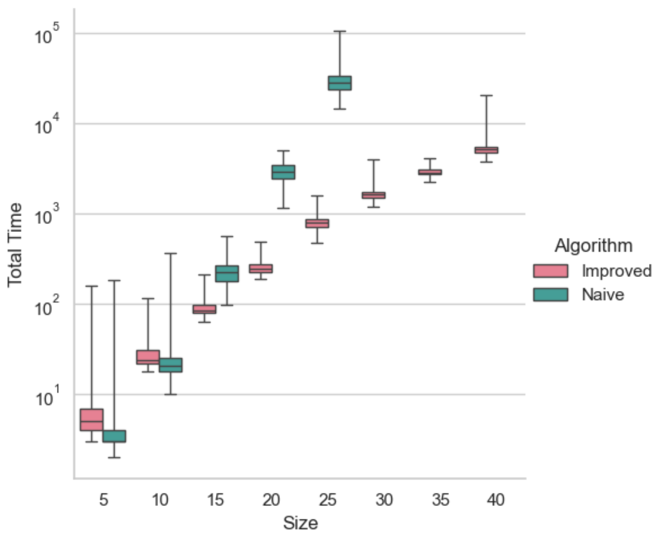


Figure 6.1: Performance results of both algorithms on puzzles of different sizes.

a puzzle using their respective strategies and we note the time it took to generate the puzzle. Then we check if the generated puzzle is uniquely solvable using the improved algorithm. To do this, we generate the CNF formula normally and then add a clause to the CNF formula to see whether another solution can be found, as described in Section 5.2

For the puzzles that are generated with our second strategy and are not uniquely solvable, we also attempt to adjust the puzzle in such a way that it becomes uniquely solvable. We call this the reduced version of the second strategy. The process for this is described in Section 5.3.

Table 6.2 shows the size of the puzzle, which strategy was used, the time it took to generate the puzzle, the time it took to check whether the puzzle is uniquely solvable, and the percentage of uniquely solvable puzzles. We used density 0.5 for the naive algorithm.

Figure 6.2 shows one graph containing the total time it took to generate and verify the uniqueness of the puzzles of different sizes using both strategies and one graph containing the amount of uniquely solvable puzzles out of 500 puzzles generated by each strategy.

We can see that the first strategy is overall slower than the second strategy, but also that it is less successful in generating uniquely solvable puzzles. It is interesting to see the difference in solving time between the two strategies. While the generated puzzles in both strategies are solved with the same algorithm, the second strategy is significantly faster in solving the puzzles.

Size	Strategy	Generation (ms)	Solving (ms)	Unique
5x5	First	4.11	0.02	77.6%
5x5	Second	8.31	0.02	91.2%
5x5	Second, reduced	7.50	1.08	93.8%
10x10	First	28.02	0.24	54.2%
10x10	Second	35.48	0.14	73.5%
10x10	Second, reduced	38.44	1.31	95.9%
15x15	First	93.69	3.56	27.0%
15x15	Second	121.35	0.82	59.2%
15x15	Second, reduced	118.2	4.38	89.1%
20x20	First	301.27	49.88	15.4%
20x20	Second	300.84	3.34	42.7%
20x20	Second, reduced	321.4	17.49	87.6%
25x25	First	875.49	297.84	5.2%
25x25	Second	889.69	6.81	29.4%
25x25	Second, reduced	915.83	50.32	83.4%
30x30	First	1284.65	1626.93	3.4%
30x30	Second	1323.4	11.65	19.0%
30x30	Second, reduced	1128.25	110.33	72.6%
35x35	First	2413.2	10020.04	0.8%
35x35	Second	2166.94	14.44	7.9%
35x35	Second, reduced	2014.88	235.57	63.9%
40x40	First	3638.45	22190.12	0.0%
40x40	Second	3523.75	23.49	3.4%
40x40	Second, reduced	3549.5	767.03	56.6%

Table 6.2: Performance results of both strategies generating puzzles of different sizes.

Table 6.3 shows the results of the reduction of puzzles that are not uniquely solvable. We can see that the algorithm is succesfull in making most puzzles uniquely solvable and also within a reasonable amount of time. The reduction time also grows at approximately the same rate as the number of solutions. This makes sense as the more solutions a puzzle has, the higher the number of variable fields is and thus the higher the number of fields involved in the reduction process.

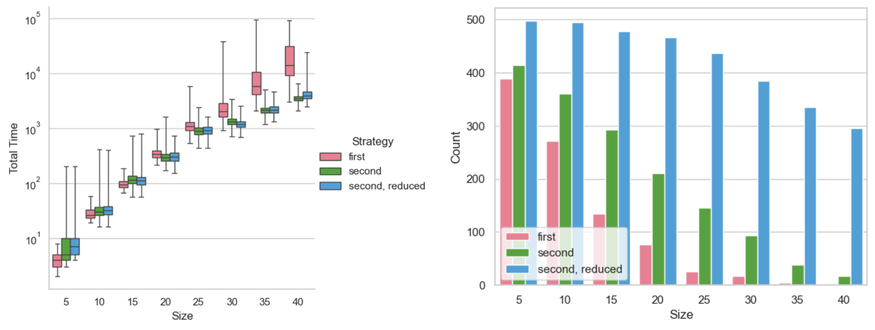


Figure 6.2: Performance results of both strategies generating puzzles of different sizes.

Size	Reduced puzzles	Reduction (ms)	Solutions	Success rate
5x5	48	1.08	2	93.8%
10x10	121	1.07	3	95.9%
15x15	201	1.37	4	89.1%
20x20	274	7.30	35	87.6%
25x25	379	26.08	98	83.4%
30x30	420	126.61	155	72.6%
35x35	457	526.55	205	63.9%
40x40	472	1013.40	238	56.6%

Table 6.3: Performance results of reduced puzzles.

6.3 Downloaded versus Generated puzzles

Now that we have measured the performance of the generation algorithms, we can compare the performance of solving puzzles generated by the generation algorithms versus puzzles downloaded from Simon Tatham's Portable Puzzle Collection. To do this, we take all of the uniquely solvable puzzles generated by the generation algorithms and all of the downloaded puzzles and let the improved algorithm solve them. In Table 6.4 and Figure 6.3 we can see the results of this experiment. We can see that the total time for generating and solving puzzles is around the same for both generated and downloaded puzzles, but downloaded puzzles take a bit longer. We do see that the downloaded puzzles take longer to generate the CNF formula, but how the solving time compares differs per size. In general, we can say that the downloaded puzzles are slightly more difficult to solve than the generated puzzles.

Size	Method	Generation (ms)	Solving (ms)
5x5	Downloaded	6.08	0.05
5x5	Generated, first	6.84	0.10
5x5	Generated, second	5.74	0.07
5x5	Generated, second, reduced	5.74	0.07
10x10	Downloaded	26.63	0.14
10x10	Generated, first	23.40	0.24
10x10	Generated, second	19.93	0.12
10x10	Generated, second, reduced	20.05	0.21
15x15	Downloaded	90.37	0.98
15x15	Generated, first	81.36	2.46
15x15	Generated, second	71.99	0.89
15x15	Generated, second, reduced	73.47	1.11
20x20	Downloaded	249.50	6.24
20x20	Generated, first	194.94	19.35
20x20	Generated, second	186.13	4.77
20x20	Generated, second, reduced	187.97	6.02
25x25	Downloaded	770.78	34.87
25x25	Generated, first	532.77	124.50
25x25	Generated, second	503.24	18.58
25x25	Generated, second, reduced	553.71	25.34
30x30	Downloaded	1515.35	129.89
30x30	Generated, first	1006.35	518.71
30x30	Generated, second	1012.39	72.02
30x30	Generated, second, reduced	1078.12	81.66
35x35	Downloaded	2562.76	351.56
35x35	Generated, first	1595.50	2501.50
35x35	Generated, second	1681.44	137.38
35x35	Generated, second, reduced	1891.51	228.59
40x40	Downloaded	4283.21	937.39
40x40	Generated, first	-	-
40x40	Generated, second	2776.59	423.06
40x40	Generated, second, reduced	3164.80	717.53

Table 6.4: Performance results of solving generated puzzles versus downloaded puzzles.

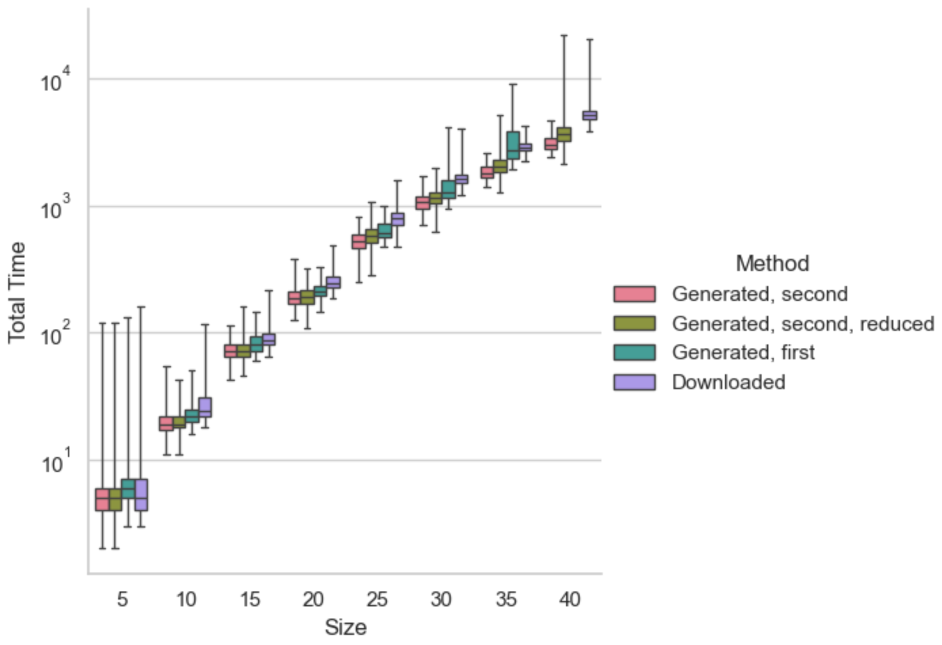


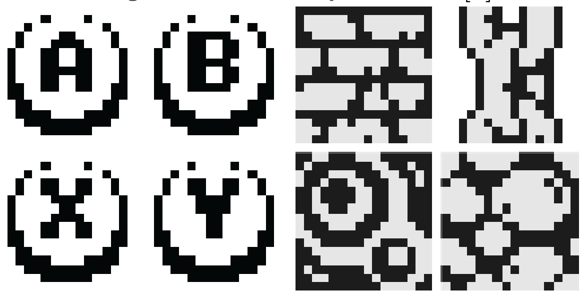
Figure 6.3: Performance results of solving generated puzzles versus downloaded puzzles.

6.4 Pixel art

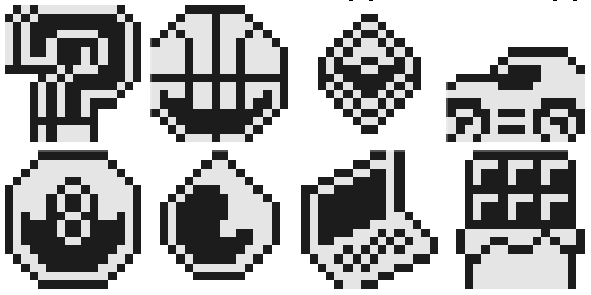
Pattern puzzles are a type of puzzle that can be seen as a form of pixel art. It is very common that in puzzle books, these puzzles create some image when solved. Therefore we also want to look at what happens when we give our solver several pixel art images. We want to find out whether there are premade images that can be converted into a uniquely solvable puzzle. We found five different sets of 16 by 16 pixels images containing a total of 1256 images. Figure 6.4 shows a few examples of each set that we used.

We wrote a simple Python script that first filtered out any illegal puzzles, and then converted each image to a text file where characters represent the color of the fields. From there we processed the text files in our Java code into a grid and then we could generate the clues for the puzzle. We then let the improved algorithm solve the puzzles, noted the time it took to solve the puzzles, and tested whether the puzzles were uniquely solvable. The results of this are in Table 6.5 and Figure 6.5.

Images from the Keyboard set[3].



Images from the Controller[3] and Tiles set[5].



Images from the Icons 1 set [1] and Icons 2 set[2].

Figure 6.4: Performance results of solving pixel art.

Set	Amount	Generation (ms)	Solving (ms)	Unique
Controller	60	121.72	1.08	68.33%
Keyboard	74	198.12	0.88	0.0%
Icons 1	441	101.51	0.68	62.59%
Icons 2	441	87.28	0.49	67.57%
Tiles	240	104.89	9.38	55.0%

Table 6.5: Performance results of solving pixel art.

We can see that the puzzles are solved in a reasonable amount of time and that a good amount of the puzzles are uniquely solvable. Only from the Keyboard set of images, there was not a single uniquely solvable puzzle. This is because the images in this set contain very few black fields, which makes it very unlikely for the puzzle to be uniquely solvable.

We also compared the performance of solving pixel art puzzles versus the performance of solving downloaded puzzles to see whether the pixel art puzzles are of similar difficulty. The results of this are in Figure 6.6. We can see that the downloaded puzzles are the same difficulty as or more difficult

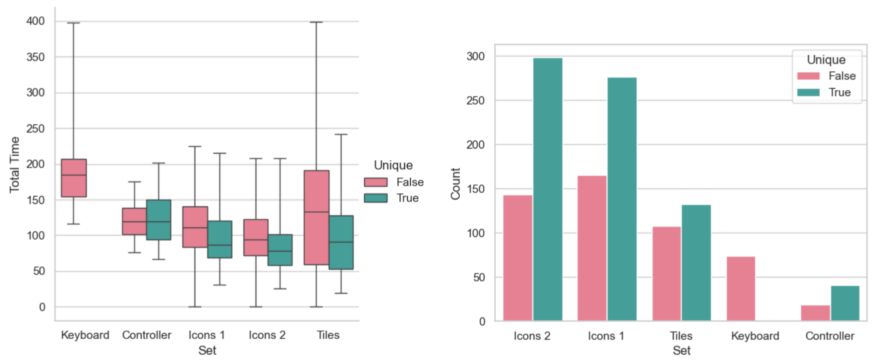


Figure 6.5: Performance results of solving pixel art.

than the pixel art puzzles. The Controller set is more difficult than the other sets, but the other sets have more difficult puzzles, but also puzzles that are less difficult than the downloaded puzzles.

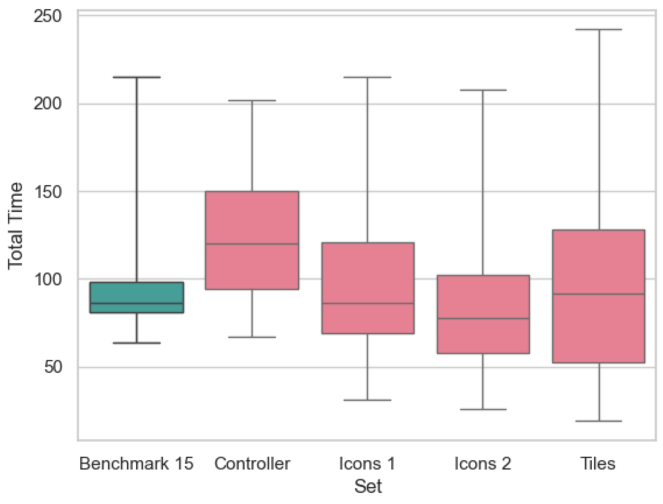


Figure 6.6: Performance results of solving pixel art(Controller, Icons 1 Set, Icons 2 Set, Tiles) compared to solving downloaded puzzles(Benchmark 15).

Chapter 7

Related Work

The Pattern puzzle has already been approached as an SAT problem[6]. However, the approach we take in this thesis is different from the ones taken in the previous work. K Joost Batenburg and Walter A Kosters took an approach using concepts of dynamic programming and network flows, whereas we take a more SAT-based approach focussing more on the constraints of the puzzle.

There is a general interest in reducing problems to the SAT problem since SAT solvers can be a very powerful tool to solve difficult problems. There is so much interest that annual SAT competitions [1] are held.

Besides the interest in reducing problems to SAT, there is also a lot of interest in logic puzzles. Various logic puzzles have been approached as SAT problems, such as the famous Sudoku puzzle[10][13], but also other puzzles like Flood-it[16], Mosaic[8], and Bridges[14]. Pattern is somewhat comparable to the Mosaic puzzle, as both of their concepts are based on coloring cells in a grid. However, the Mosaic puzzle contains an extra layer of complexity, as clues can be added or removed to make the puzzle more or less difficult. Pattern is rather limited to the constraints of the puzzle itself, which makes it a more straightforward puzzle to solve.

[1]https://satcompetition.github.io

Chapter 8

Conclusions and Future Work

In this thesis, we have shown two ways of encoding the Pattern puzzle as a SAT problem. Firstly, we showed a naive encoding, which is a brute-force approach to solving the puzzle. Secondly, we showed an improved encoding, which is based on the rules of the puzzle and the logical implications that arise from these rules.

After showing the two encodings, we have shown how to generate Pattern puzzles. We have shown two different ways of generating puzzles; one that generates puzzles by generating a random grid of black and white cells, and one that generates puzzles using a line-by-line technique that is also based on statistics found from pre-made puzzles. In the second approach, we have also shown a technique to make non-unique puzzles unique.

In our experiments, we saw that the naive encoding is not very efficient, especially for larger puzzles. The improved encoding is much more efficient. We see that both encodings scale exponentially with the size of the puzzle, but the improved encoding scales better than the naive encoding. We also ran experiments on the generation algorithms, and we saw that the algorithms do not differ much in performance for smaller puzzles. However, for larger puzzles, there is a clear difference between the first algorithm and the second algorithm. The first algorithm is much slower than the second algorithm. We then compared our generated puzzles to the downloaded puzzles, and we found that the downloaded puzzles were slightly more difficult than the generated puzzles.

Finally, we experimented with pixel art. Pattern puzzles found in puzzle books are often pixel art, and we wanted to see if we could take random black-and-white pixel art and convert it to a Pattern puzzle. We found that this is possible, but it depends on the image. Generally, images with more black pixels have a higher chance of being converted to a Pattern puzzle. How difficult these puzzles are, really depends on the image. Some are more difficult than others, but generally, they are more difficult than the downloaded puzzles.

In terms of future research, there is not much more to be done or optimized in the first encoding, as it is a brute-force approach. However, the second encoding does have the potential for further optimization. The encoding could be optimized by taking a look at the constraints of the puzzle and starting from scratch, or taking the current encoding and trying to optimize it. This could be done by trying to find implications that arise from the rules described in Section 4.4.

Furthermore, there could be more research into the generation of puzzles. Specifically on generating puzzles using different approaches. One suggestion would be to let the color of a field be dependent on the color of the fields around it. This way, the puzzle could be generated more cohesively, meaning that there are larger fields of connected black cells. Currently, we see that the generated puzzles are often very fragmented, but the downloaded puzzles seem to be more cohesive. This might also affect the difficulty of the puzzle, possibly making them slightly more difficult.

Besides this different approach to generating puzzles, there could also be research into improving the current generation algorithms. We saw that for larger puzzles, the algorithms do not generate larger puzzles as well as they do for smaller puzzles. By improving the algorithms, we could generate larger puzzles more accurately. A suggestion for this would be to take a look at the way non-unique puzzles are altered to be unique. The current approach could be extended to work better for puzzles that have a lot of variable fields.

Bibliography

[1] Icons 1 set. `https://piiixl.itch.io/1-bit-16px-icons-part-1`.

[2] Icons 2 set. `https://piiixl.itch.io/1-bit-icons-part-2`.

[3] Keyboard and controller set. `https://ansdor.itch.io/button-icons`.

[4] Pattern, from simon tatham's portable puzzle collection. `https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/patterm.html`.

[5] Tiles set. `https://piiixl.itch.io/1-bit-patterns-and-tiles`.

[6] K Joost Batenburg and Walter A Kosters. A reasoning framework for solving nonograms. In *International Workshop on Combinatorial Image Analysis*, pages 372–383. Springer, 2008.

[7] Stephen A Cook. The complexity of theorem-proving procedures. In *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, pages 143–152. 2023.

[8] Thijs de Jong, CLM Kop, and JSL Junges. Mosaic as a SAT problem. 2023.

[9] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers 6*, pages 502–518. Springer, 2004.

[10] Inês Lynce and Joël Ouaknine. Sudoku as a sat problem. In *AI&M*, 2006.

[11] RA Oosterman. *Complexity and solvability of Nonogram puzzles.* PhD thesis, Faculty of Science and Engineering, 2017.

[12] Thomas J Schaefer. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 216–226, 1978.

[13] Helmut Simonis. Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, volume 12, pages 13–27. Citeseer, 2005.

[14] Rico te Wechel, CLM Kop, and JSL Junges. Bridges as an SMT problem. 2023.

[15] G.S. Tseytin. On the complexity on derivation in propositional calculus. In *Presented at the Leningrad Seminar on Mathematical Logic*, 1966.

[16] Milan van Stiphout, CLM Kop, and H Zantema. Flood-it as a SAT problem. 2023.