Bachelor thesis
Computing Science

Radboud University

Mosaic as a SAT problem

Author:
Thijs de Jong
s1015438

First supervisor/assessor:
dr. C.L.M. Kop
C.Kop@cs.ru.nl

Second assessor:
dr. J.S.L. Junges
sebastian.junges@ru.nl

March 24, 2023

Abstract

Mosaic is a logic puzzle that has been proven to be an NP-Complete problem. This means that the problem is both in NP and NP-Hard. In this thesis, our aim is to encode the Mosaic puzzle as a Boolean Satisfiability (SAT) problem in two different ways and solve it using a SAT solver. Using this, an efficient way to generate new Mosaic puzzles will be developed. We will test the performance of both encodings, as well as try to optimize one of them. We also look into optimizing the generation method. We find no improvement to the original methods.

Contents

Chapter 1

Introduction

Logical puzzles have been around for quite a while now. They were first introduced in the late 19^{th} century by Lewis Caroll [17]. Some of the most well-known logical puzzles include Sudoku and Master-Mind, while there is an extensive list of logical puzzles that are less known to the public, like Flood, Towers, and Mosaic. Even though the puzzles look nothing alike, they all share the same characteristics; the puzzles are fairly easy to understand and have a small set of rules and restrictions, but the difficulty to solve them can range from very easy to extremely difficult.

Logical puzzles like Sudoku and Mosaic have been proven to be NP-Complete [15] [10]. NP-Complete problems are defined by two rules. The first rule is that problem p is in NP, or the Non-deterministic Polynomial time complexity class. This set contains problems where solutions can be verified in polynomial time. The second rule is that problem p is in NP-Hard, meaning that every problem in NP is reducible to problem p in polynomial time: problem p is at least as hard as the hardest problem in NP.

NP-Complete problems can be reduced to other NP-Complete problems. This means that solving certain problems in the NP-Complete class extremely fast can be useful for other NP-Complete problems, or even applied to real life problems. In this thesis, we will reduce the Mosaic logic puzzle to another NP-Complete problem, namely the SAT problem, which is known as the first problem proven to be NP-Complete [7].

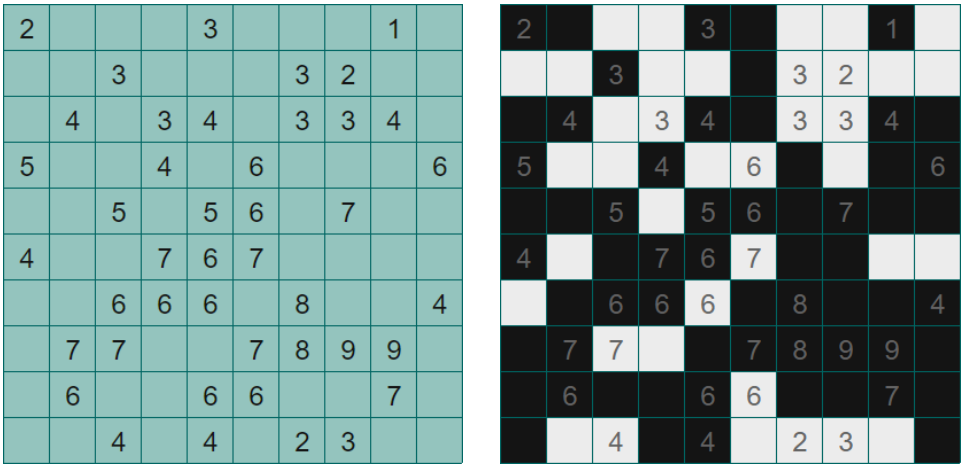
We will start by reviewing some preliminary knowledge on the rules of Mosaic and boolean formulas (Chapter 2). Next, we will discuss a naive encoding for translating the Mosaic logic puzzle to a SAT problem (Chapter 3) and a way to generate puzzles (Chapter 4). After that, we will look into a better solution for encoding the Mosaic logic puzzle (Chapter 5). Then, we will compare both encodings, and look into other factors that might speed up the encoding (Chapter 6). To complete, we will compare our work with existing work (Chapter 7), summarize the research, and look at future research (Chapter 8).

Chapter 2

Preliminary Knowledge

2.1 Mosaic

Mosaic is a logic puzzle wherein the player must reveal a picture in the style of pixel art by using clues to find out what 'pixels' to colour. It consists of a grid of any size, where the cells in the grid are either empty or contain a number between 0 and 9. The goal of the puzzle is to find a filling pattern of black and white squares, where the colour of the cells gets decided by the numbers in the puzzle. These numbers are clues for how many black cells there are around the cell with the clue, including the cell with the clue itself, so that the number of black cells matches the clue. When all clues have the correct number of black cells around them, the puzzle is solved. Figure 2.1a shows an example of a puzzle, with the solution in figure 2.1b.



(a) (b)

Figure 2.1: A full-size example of an unsolved Mosaic, with the solution next to it

For a Mosaic to be valid, it needs to be uniquely solvable. In other words, there should be only one solution possible. This implies that clues cannot contradict each other, as no solution will be possible. Next to that, this also means that every cell must contain a clue, or there must be at least one clue in the surrounding cells. If this is not the case, there would be multiple solutions possible. A simple example of solving a Mosaic puzzle is shown in figure 2.2.

The puzzle goes by other names, like Fill-A-Pix, as implemented by ConceptisPuzzles [1], who adopted this puzzle, after Trevor Truran invented the puzzle in the 1980s, based on Conway's Game of Life [2]. Fill-A-Pix, and thus Mosaic, is an NP-Complete puzzle [10].

Figure 2.2: A step-by-step example of solving a Mosaic puzzle.

2.2 Normal forms in boolean logic

In boolean logic, formulas are created with variables that can either be true or false, and operations that link those variables. The three main operations are the conjunction, or *AND*, the disjunction, or *OR*, and the negation, or *NOT*. The symbols used are $\wedge$, $\vee$, and $\neg$ respectively. With the variables and operations, boolean formulas can be created.

Since boolean formulas can be very complex, they can be rewritten in a way that is easier to work with. When a boolean formula is rewritten in a way it can not be rewritten any further, it is called a Normal Form. Depending on how the formula gets rewritten, the Normal Form can be different. There are two important normal forms; the Conjunctive Normal Form (CNF), and the Disjunctive Normal Form (DNF).

A CNF formula is a formula where there are conjunctions between all clauses, and the clauses exist of disjunctions of literals. A literal is a variable or a negated variable. An example of a CNF formula is:

$$A \wedge (B \vee \neg C) \wedge (D \vee E \vee F) \tag{2.3}$$

A DNF formula is the opposite of a CNF formula, namely a formula where there are disjunctions between all clauses, and the clauses consist of conjunctions of literals. A literal is still a variable or a negated variable. An example of a DNF formula is:

$$(A \wedge B) \vee (B \wedge C \wedge \neg D) \vee E \tag{2.4}$$

Every formula in boolean logic can be converted to an equivalent formula that is in CNF or DNF. The conversion to CNF or DNF can be done by using logical equivalences [8]. Logical equivalences are achieved when two formulas have the same truth value in every model but are not the same formula. This is denoted as $p \equiv q$.

For Mosaic, we will need to convert DNF formulas to CNF formulas. To achieve this, the following logical equivalences are needed:

Name		**Equivalence**
Double negation law		$\neg(\neg(p)) \equiv p$
Distributive laws	1	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
	2	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
De Morgan's laws	1	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
	2	$\neg(p \vee q) \equiv \neg p \wedge \neg q$
Absorption laws	1	$p \vee (p \wedge q) \equiv p$
	2	$p \wedge (p \vee q) \equiv p$

We can apply these rules to boolean formulas to convert them to CNF formulas. We want for example convert the following formula to a CNF formula:

$$(A \wedge B) \vee (B \wedge C \wedge \neg D) \vee E \tag{2.5}$$

By applying the logical equivalences as described before, this formula will be converted to the following CNF formula:

$$\begin{aligned}(B \vee A \vee E) \wedge (C \vee A \vee E) \wedge (\neg D \vee A \vee E) \\ \wedge (B \vee E) \wedge (C \vee B \vee E) \wedge (\neg D \vee B \vee E)\end{aligned} \tag{2.6}$$

For the conversion of a formula to a CNF, we can apply the Distributive law [1] and De Morgan's law [1]. If we want to convert a formula to a DNF, we can apply the Distributive law [2] and De Morgan's law [2]. If we would use both laws for conversion to a normal form, there would never be a normal form, as we could infinitely apply a law to convert the formula.

2.3 Tseitin Transformation

Another way of converting boolean formulas to CNF is by applying the Tseitin transformation. Where transforming boolean formulas with logical equivalences can result in exponential sizes of the formula, with the Tseitin transformation, the formula size grows linearly relative to the original formula's size, and the transformation can be done in linear time [19].

The Tseitin transformation works as follows. Take a formula ϕ, and assign every subformula in ϕ to a new variable x_i, starting with the formula itself. Literals do not need to be assigned to a new variable, as they can keep their name. Then, the original formula can be expressed as follows:

$$x_1 \wedge (x_1 \equiv ...) \wedge (x_2 \equiv ...) \wedge (x_3 \equiv ...) \tag{2.7}$$

Now, every subformula can be transformed into a CNF formula, using the following conversions:

Formula	**CNF**
$p \equiv \neg q$	$(p \vee q) \wedge (\neg p \vee \neg q)$
$p \equiv q \wedge r$	$(\neg p \vee q) \wedge (\neg p \vee r) \wedge (p \vee \neg q \vee \neg r)$
$p \equiv q \vee r$	$(p \vee \neg q) \wedge (p \vee \neg r) \wedge (\neg p \vee q \vee r)$
$p \equiv q \rightarrow r$	$(p \vee q) \wedge (p \vee \neg r) \wedge (\neg p \vee \neg q \vee r)$

Example 2.3.1. We will convert the following formula to a CNF formula

$$\phi := p \rightarrow ((q \vee r) \wedge s) \tag{2.8}$$

The subformulas of this formula will look as follows:

$$\begin{aligned} x_1 &\equiv (p \rightarrow ((q \vee r) \wedge s)) \\ x_2 &\equiv ((q \vee r) \wedge s) \\ x_3 &\equiv (q \vee r) \end{aligned} \tag{2.9}$$

Now, the original formula can be expressed as the following:

$$T(\phi) := x_1 \wedge (x_1 \equiv p \rightarrow x_2) \wedge (x_2 \equiv x_3 \wedge s) \wedge (x_3 \equiv q \vee r) \tag{2.10}$$

Rewriting this with the transformations in the table above gives:

$$\begin{aligned} T(\phi) := x_1 \\ &\wedge (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg p \vee x_2) \\ &\wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee s) \wedge (x_2 \vee \neg x_3 \vee \neg s) \\ &\wedge (x_3 \vee \neg q) \wedge (x_3 \vee \neg r) \wedge (\neg x_3 \vee q \vee r) \end{aligned} \tag{2.11}$$

Which is a CNF formula.

2.4 Solving Boolean Satisfiability Problems

The Boolean Satisfiability Problem, also known as and hereafter referred to as the SAT problem, is the problem of determining whether there exists a solution of a boolean formula where all the variables in the boolean formula are either `true` or `false`, such that the boolean formula will evaluate to `true`. If this is the case, the boolean formula is satisfiable. Otherwise, the boolean formula is unsatisfiable. SAT is the first problem known to be NP-Complete [7].

Multiple so-called SAT solvers have been developed to automatically solve SAT problems. A SAT solver takes a CNF formula as input, checks whether the formula is satisfiable, and then outputs either *unsatisfiable*, or *satisfiable* together with a solution to let the formula evaluate to `true`. Even if there are multiple solutions for a formula, the SAT solver only displays one, as its primary task is just to check whether the formula is satisfiable.

For the Mosaic puzzle, we will be using the Sat4j java library [13], which is a library that can solve SAT problems in java. Like most SAT solvers, Sat4j uses the Dimacs format as input [4]. The Dimacs format is a way of writing CNF formulas such that SAT solvers can easily process the formulas. Converting a CNF formula to the Dimacs format is fairly straightforward; all literals are written as an integer and negations are written as negative integers. One line in the Dimacs format can be seen as a clause, where all integers are connected by disjunctions. Each line ends with a 0, and all lines (clauses) are connected by conjunctions. A Dimacs file starts with the line `'p cnf <variables> <clauses>'`, where `<variables>` and `<clauses>` are replaced by the total number of variables and clauses in the CNF formula.

Example 2.4.1. Suppose we have the following CNF formula, which we want to rewrite to the Dimacs format:

$$A \wedge (B \vee \neg C) \wedge (C \vee D \vee E \vee \neg F) \tag{2.12}$$

This could then be rewritten to the following:

```
p cnf 6 3
1 0
2 -3 0
3 4 5 -6 0
```

2.5 Unit Propagation

Unit propagation, or unit clause propagation, is a way of simplifying boolean formulas that is broadly used by SAT solvers to speed up the evaluation of a formula significantly [5] [14]. The process of unit propagation uses unit clauses (clauses that consist of a single literal) in a CNF to simplify the entire CNF. Since each clause needs to be satisfied in order to let the entire CNF be satisfiable, we know that a single literal l must be satisfied. Using this information, we can simplify the rest of the CNF based on two rules:

1. every clause containing the literal l is removed, except for the unit clause itself;

2. the negated literal $\neg l$ is removed in every clause.

Rule 1 applies since a clause consists of a disjunction of literals, and since literal l is already satisfied, the entire clause is satisfied.

Rule 2 applies since the negated literal $\neg l$ can never contribute to the clause being satisfied, so there is no need to evaluate it anymore.

The same applies for a negated literal $\neg l$ as unit clause, but then every clause containing the literal $\neg l$ is removed, and the literal l is removed from every clause, since the negated negated literal is the literal itself again.

Example 2.5.1. Suppose the following CNF where we want to apply unit propagation to simplify the CNF:

$$(\neg A \vee B) \wedge (A \vee C) \wedge (B \vee \neg C) \wedge A \tag{2.13}$$

We have the unit clause A, so unit propagation can be applied to simplify this CNF. The clause $(A \vee C)$ can be removed altogether according to rule 1. In the first clause $(\neg A \vee B)$, we can remove $\neg A$ according to rule 2. The resulting CNF will be:

$$B \wedge (B \vee \neg C) \wedge A \tag{2.14}$$

Now, a new unit clause is introduced, namely B, meaning we can apply unit propagation once more to remove the clause $(B \vee \neg C)$, according to rule 1. We then end up with the following CNF:

$$B \wedge A \tag{2.15}$$

Chapter 3

Naive solution to the SAT problem of Mosaic

We will use a SAT solver to solve Mosaic, as described in chapter 2.4. To be able to do this, we need to convert the puzzle to a CNF formula, such that it can be solvable with a SAT solver. In this chapter, we will look into a way to encode Mosaic puzzles as a SAT problem in order to convert it into a CNF formula, to solve it using a SAT solver.

In order to achieve this, we divide the problem into three steps. The first step is to find a viable method to encode the Mosaic puzzle into a boolean formula. Next, we can use the Tseitin transformation to convert this formula to a CNF formula. This CNF formula can then be used as input for a SAT solver. The SAT solver outputs a solution as a string if the CNF formula is satisfiable. This solution can then be converted back to a solution of the Mosaic puzzle.

3.1 Encoding Mosaic into a boolean formula

To be able to encode the Mosaic into a boolean formula, we will have to break down the Mosaic. A Mosaic consists of a grid with width w and height h, having a total of $w*h$ cells. Usually, the height and the width have the same value, but this does not have to be the case. Cells have an x and a y coordinate. For each cell, the cell can contain a value v, telling us how many black cells there should be around this cell, including itself. This value has to be between -1 and 9, where -1 means that the cell does not contain a clue. In the case that there is no clue, the number of black cells surrounding this cell can only be determined by looking at clues in the surrounding cells. A cell has up to 9 neighbours, including itself.

To encode the Mosaic to a boolean formula, we define a variable for every cell in the Mosaic puzzle. A cell variable can be defined as $c_{x,y}$, where x and y are the coordinates of the respective cell. For instance, a cell on

coordinates `(3,4)` would be represented as $c_{3,4}$. If $c_{3,4}$ evaluates to `true` by the SAT solver, the cell linked to the variable $c_{3,4}$, which has coordinates `(3,4)`, should be black in order to achieve a solution. If $c_{3,4}$ evaluates to `false`, the linked cell should be white. We can use this later on to convert the valuation of the SAT solver to a solution.

To create a boolean formula for the entire grid, we have to look at a single cell, encode the clue of this cell as a boolean formula over the cell variables, repeat this for every cell containing a clue, and append all the formulas together. In order to find the boolean formula for one cell, we first look at the surrounding cells of this particular cell and store those in an array. This includes the cell itself. This is done by checking in every possible direction (orthogonal and diagonal) whether there is a cell.

Using the surrounding cells, we can create a boolean formula for the cell. Since the clue of the cell equals the number of surrounding cells that should be black, the easiest way to make a boolean formula that evaluates which cells should be black in order to find a solution is by listing all possible combinations of the surrounding cells, where there are as many black cells as the clue value, and the remaining cells are white. In the encoding, cells can be denoted as their respective variable, where black cells are just the variable, and white cells are the negated variable.

Example 3.1.1. We will use figure 3.1 as an example. We will look at the cell with coordinates `(1,0)`, which has variable $c_{1,0}$. In the figure, this cell is coloured red. The array of the variables of the surrounding cells is $\{c_{0,0}, c_{1,0}, c_{2,0}, c_{0,1}, c_{1,1}, c_{2,1}\}$. The clue of the cell is 2, so we need to find all possible combinations of the surrounding cells, where 2 of the cells are black and 4 of the cells are white. All of these possible combinations can be seen in figure 3.3.

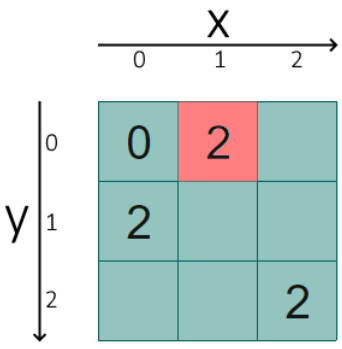


Figure 3.1: An example configuration of a Mosaic puzzle

If we put all these combinations together, we get the following boolean formula:

$$
\begin{aligned}
&(c_{0,0} \wedge c_{1,0} \wedge \neg c_{2,0} \wedge \neg c_{0,1} \wedge \neg c_{1,1} \wedge \neg c_{2,1}) \vee \\
&(c_{0,0} \wedge \neg c_{1,0} \wedge c_{2,0} \wedge \neg c_{0,1} \wedge \neg c_{1,1} \wedge \neg c_{2,1}) \vee \\
&(c_{0,0} \wedge \neg c_{1,0} \wedge \neg c_{2,0} \wedge c_{0,1} \wedge \neg c_{1,1} \wedge \neg c_{2,1}) \vee \\
&(c_{0,0} \wedge \neg c_{1,0} \wedge \neg c_{2,0} \wedge \neg c_{0,1} \wedge c_{1,1} \wedge \neg c_{2,1}) \vee \\
&(c_{0,0} \wedge \neg c_{1,0} \wedge \neg c_{2,0} \wedge \neg c_{0,1} \wedge \neg c_{1,1} \wedge c_{2,1}) \vee \\
&(\neg c_{0,0} \wedge c_{1,0} \wedge c_{2,0} \wedge \neg c_{0,1} \wedge \neg c_{1,1} \wedge \neg c_{2,1}) \vee \\
&(\neg c_{0,0} \wedge c_{1,0} \wedge \neg c_{2,0} \wedge c_{0,1} \wedge \neg c_{1,1} \wedge \neg c_{2,1}) \vee \\
&(\neg c_{0,0} \wedge c_{1,0} \wedge \neg c_{2,0} \wedge \neg c_{0,1} \wedge c_{1,1} \wedge \neg c_{2,1}) \vee \\
&(\neg c_{0,0} \wedge c_{1,0} \wedge \neg c_{2,0} \wedge \neg c_{0,1} \wedge \neg c_{1,1} \wedge c_{2,1}) \vee \\
&(\neg c_{0,0} \wedge \neg c_{1,0} \wedge c_{2,0} \wedge c_{0,1} \wedge \neg c_{1,1} \wedge \neg c_{2,1}) \vee \\
&(\neg c_{0,0} \wedge \neg c_{1,0} \wedge c_{2,0} \wedge \neg c_{0,1} \wedge c_{1,1} \wedge \neg c_{2,1}) \vee \\
&(\neg c_{0,0} \wedge \neg c_{1,0} \wedge c_{2,0} \wedge \neg c_{0,1} \wedge \neg c_{1,1} \wedge c_{2,1}) \vee \\
&(\neg c_{0,0} \wedge \neg c_{1,0} \wedge \neg c_{2,0} \wedge c_{0,1} \wedge c_{1,1} \wedge \neg c_{2,1}) \vee \\
&(\neg c_{0,0} \wedge \neg c_{1,0} \wedge \neg c_{2,0} \wedge c_{0,1} \wedge \neg c_{1,1} \wedge c_{2,1}) \vee \\
&(\neg c_{0,0} \wedge \neg c_{1,0} \wedge \neg c_{2,0} \wedge \neg c_{0,1} \wedge c_{1,1} \wedge c_{2,1})
\end{aligned}
\tag{3.2}
$$

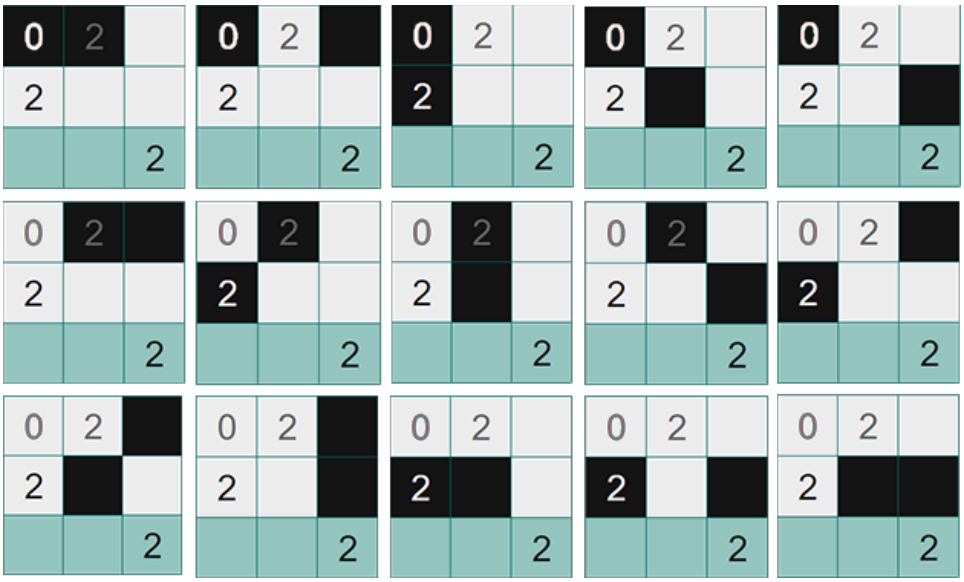


Figure 3.3: All possible solutions for the cell on coordinates (1,0), with variable $c_{1,0}$ and clue 2

We now have a boolean formula of one cell. The boolean formula needs to be converted to a CNF formula to be able to rewrite it to Dimacs format and use it in a SAT solver.

3.2 Converting the encoding to a CNF

There are two ways of converting a boolean formula to a CNF, as explained in chapter 2.2. The first way is by using logical equivalences, but using this would result in an exponential growth in equation size. Instead, we will be using the Tseitin transformation, as it outputs a formula whose size grows linearly, because it makes use of auxiliary variables [6]. The Tseitin transformation is explained in chapter 2.3. If we apply the Tseitin transformation to the boolean formula, we will get as output a CNF formula.

Lastly, we can repeat this way of encoding a cell for all cells in a Mosaic puzzle. To achieve this, we will loop through every cell in the puzzle. If the cell contains a clue, we can create an array with variables of the surrounding cells, list all the possible solutions, and convert the resulting boolean formula to a CNF using the Tseitin transformation. The resulting CNF will be added to the general encoding for the puzzle. Concatenations of CNFs can be arranged by using a conjunction. This way, a concatenation of CNFs will stay a CNF.

3.3 Creating a solution

Now that we have a CNF formula for a Mosaic puzzle, we can rewrite the entire CNF formula to the Dimacs format. This Dimacs file can be entered into a SAT solver. If the given CNF formula is satisfiable, the SAT solver returns a list of positive and negative numbers corresponding to the variables. The list shows a valuation to the entered CNF formula, which can be seen as a solution to the given Mosaic puzzle. For instance, if the returned list is `(1,-2,-3,5)`, the variables `1` and `5` have been evaluated to `true`, `2` and `3` have been evaluated to `false`, and variable `4` can be either `true` or `false`. Since the variables correspond with the cells in the Mosaic puzzle, we can easily see which cells should be black, and which cells should be white.

In order to confirm that the SAT solver is correct, 35 different configurations have been downloaded from Simon Tatham's Portable Puzzle Collection [3], and the answer that the SAT solver came up with has been checked with non-SAT code whether all the clues comply to the requirements. Those test configurations were of all sizes, starting at 5x5 and going up to 100x100.

Chapter 4

Generating puzzles

4.1 Checking if a puzzle has a unique solution

In order to be able to generate uniquely solvable puzzles, we need to be able to check if a given configuration of a puzzle is unique. This can be done by adding the boolean formula that states that the known solution is not a possible solution anymore for this configuration to the encoding of the mosaic puzzle. This can be seen as 'blocking' the actual solution from being a solution. With this, we can check whether the current configuration of the game is uniquely solvable. If the CNF is satisfiable, there is another solution for the game, making it nonunique. This can be done at any given point in the generation process.

Example 4.1.1. Suppose we have a simple original configuration, shown in figure 4.1, for which we want to check whether it is unique.

0	0	1	2	2
2	2	3	3	3
3	4	3	4	3
5	6	4	3	2
3	4	2	2	1

Figure 4.1: Simple starting configuration of a Mosaic

To do so, we need to add the boolean formula that states that the current correct solution is not possible anymore as a solution. This way, if the SAT solver evaluates the encoded puzzle to `unsatisfiable`, there is no other solution, and the configuration of the puzzle is unique. If the SAT solver evaluates the encoded puzzle to `satisfiable`, together with a list of

which variables have to be true and which false, there is another solution next to the known solution. The solution of the given configuration can be written down as the following CNF:

$$\begin{aligned} &\neg c_{0,0} \wedge \neg c_{1,0} \wedge \neg c_{2,0} \wedge c_{3,0} \wedge \neg c_{4,0} \wedge \\ &\neg c_{0,1} \wedge \neg c_{1,1} \wedge \neg c_{2,1} \wedge \neg c_{3,1} \wedge c_{4,1} \wedge \\ &c_{0,2} \wedge c_{1,2} \wedge \neg c_{2,2} \wedge c_{3,2} \wedge \neg c_{4,2} \wedge \\ &c_{0,3} \wedge \neg c_{1,3} \wedge c_{2,3} \wedge \neg c_{3,3} \wedge c_{4,3} \wedge \\ &c_{0,4} \wedge c_{1,4} \wedge \neg c_{2,4} \wedge \neg c_{3,4} \wedge \neg c_{4,4} \end{aligned} \tag{4.2}$$

Here, $c_{x,y}$ is a variable linked to the corresponding cell on coordinates `(x,y)` in the 5x5 grid. If the SAT solver evaluates this CNF to true, it represents the solution that can be seen in figure 4.1. We now add this solution to the CNF, but in a way that it will never be possible to be the solution. This can be done by negating boolean formula 4.2:

$$\neg(\neg c_{0,0} \wedge \neg c_{1,0} \wedge \cdots \wedge c_{1,2} \wedge \neg c_{2,2} \wedge c_{3,2} \wedge \cdots \wedge \neg c_{3,4} \wedge \neg c_{4,4}) \tag{4.3}$$

To add this boolean formula to the encoding, it needs to be a clause for a CNF. We can transform this boolean formula to a formula with only disjunctions by applying De Morgan's law and the double negation law, which are introduced in chapter 2.2:

$$(c_{0,0} \vee c_{1,0} \vee \cdots \vee \neg c_{1,2} \vee c_{2,2} \vee \neg c_{3,2} \vee \cdots \vee c_{3,4} \vee c_{4,4}) \tag{4.4}$$

We can now append this clause to our encoding, so that the unique solution will not be possible as a solution anymore.

4.2 Generating puzzles using the SAT encoding

In order to generate a puzzle, some steps have to be taken. First, a grid has to be generated of any chosen size, where every cell in the grid is either black or white. This will be determined randomly. After that, for every cell, the correct clue value will be filled in, depending on the number of black cells surrounding the cell, including the cell itself. Once the entire grid is filled with clues, we first check whether the original solution is unique using the method described in chapter 4.1. If this is not the case, we will regenerate the starting grid and the corresponding clues, until we have found a unique original solution.

We can now check for every cell whether the clue is important for a unique solution, so that we can remove clues in order to get a puzzle with the least number of clues needed. For every cell in the puzzle, we will try to remove the clue in the cell from the puzzle, and try to evaluate the encoding of the puzzle with the SAT solver again. If the SAT solver can't find a

solution, the puzzle is still uniquely solvable, since the correct solution is blocked, so the clue will stay removed. If it can find a solution, the clue will be put back in the puzzle, as this clue is important for making the puzzle uniquely solvable. This will be done for every cell in the puzzle, going from left to right, from top to bottom. When for every cell it has been checked whether the clue is important for the unique solutions, the result is a configuration of the Mosaic puzzle that is uniquely solvable.

Example 4.2.1. In this example, we will show the steps taken to generate a Mosaic puzzle. The size of the puzzle that will be generated is 5x5. To start with, we will set every cell in the 5x5 grid to black or white randomly. The result can be seen in figure 4.5.

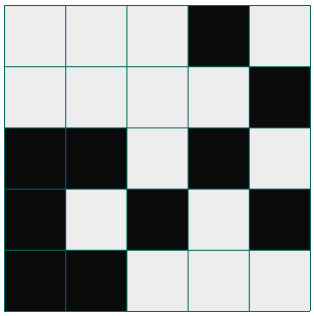


Figure 4.5: Setting all cells randomly to either black or white

After this is done, we will fill every cell in the 5x5 grid with its corresponding clue. This can be done by counting the number of black cells surrounding the cell, including the cell itself. The result of this can be seen in figure 4.6.

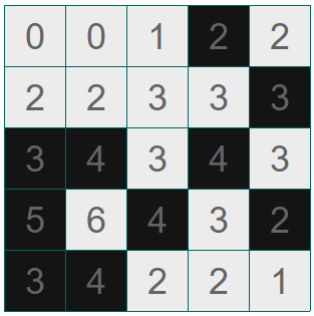
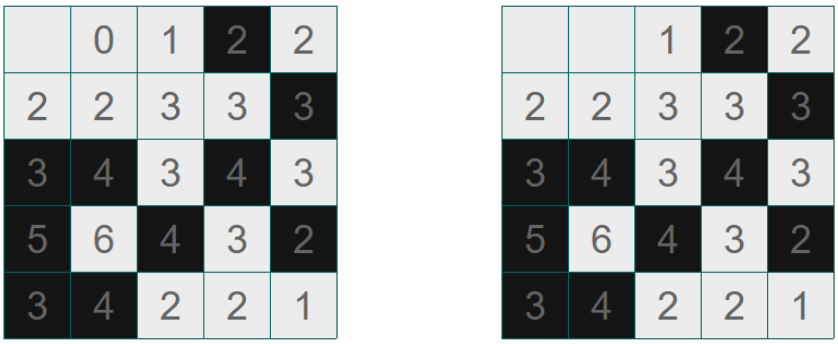


Figure 4.6: Computing the clues for all cells

Now that we have a basic configuration of the Mosaic, we first check whether this original configuration is unique according to chapter 4.1, which is the case. After this, we can start with the removal of the clues that are

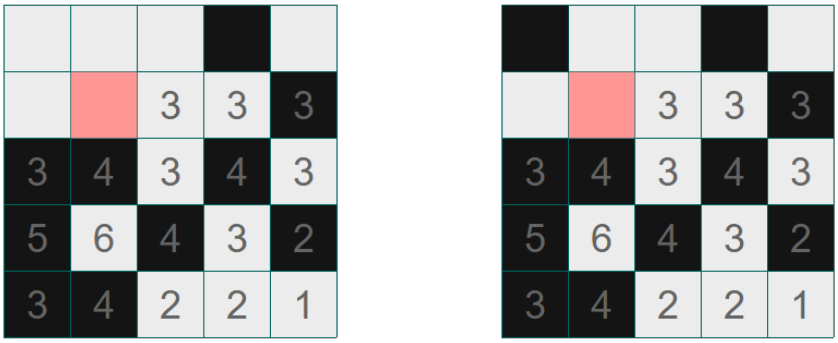
not necessary for a unique solution. We will try to remove clues one by one, convert the new configuration to an encoding, and try to solve this encoding with a SAT solver. This will be done from left to right, top to bottom. The clue of the first cell with coordinates `(0,0)` can be removed, as can be seen in figure 4.7a, since without this clue, the configuration is still uniquely solvable, as the SAT solver evaluates to `unsatisfiable`. The same goes for the second cell, with coordinates `(1,0)`, as can be seen in figure 4.7b.



(a) (b)

Figure 4.7: The first clues can be removed since the new configuration is still uniquely solvable without them

We can continue this removal of clues until the 7^{th} cell. At this cell, with coordinates `(1,1)`, we can see that upon removal, a solution other than the original solution is possible. The original solution can be seen in figure 4.8a, and the new solution, found by the SAT solver, in figure 4.8b. The red cell is the cell from which the clue got removed. The top left cell, on coordinates `(0,0)`, can be either black or white, without interfering with any clue, meaning that the clue in the cell on coordinates `(1,1)` is important for a unique solution. We can continue the removal of clues this way to end up with a uniquely solvable configuration of a Mosaic puzzle, which can be seen in figure 4.9.



(a) (b)

Figure 4.8: Two different solutions are possible upon removal of the clue in the cell indicated with red

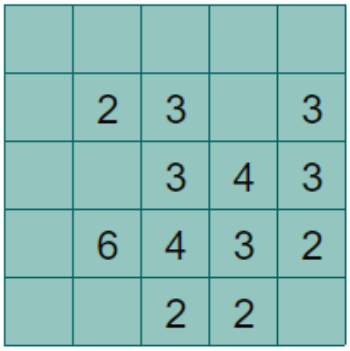


Figure 4.9: A uniquely solvable generated configuration of a Mosaic puzzle

Chapter 5

A better solution to the SAT problem

5.1 Why we need a better encoding

In chapter 3, we have explained how SAT solvers work, and how they are used to solve the Mosaic puzzle. However, the encoding we have used is not the best and can be improved upon. In this chapter, we will explain why we need a better encoding and how we can improve the encoding.

The problem with the current implementation is that the number of variables in one DNF grows based on a binomial distribution. This is because the number of clauses in the DNF is based on a binomial coefficient, where the problem can be seen as $\binom{n}{k}$, where n is the number of surrounding cells including the cell itself of the cell we want the DNF for, and k is the value of the clue in the cell. To simplify the problem, we have k squares of which n should be coloured black. In the case $\binom{9}{1}$, so a 3x3 grid where the clue in the middle is 1, the number of clauses in the DNF is 9, which is not that much of a problem. But when the clue in the middle is 5, so $\binom{9}{5}$, the number of clauses in the DNF is 126, which is already a lot. After this step, the DNF still needs to be converted to a CNF using the Tseitin transformation, which causes the number of clauses to grow to 1145 for every such clue, making the encoding very slow.

The solution to this problem is to find an encoding which makes use of auxiliary variables to reduce the number of clauses in the CNF. Instead of trying to list all the possible combinations of black squares, we can use auxiliary variables to make sure that the number of black squares is equal to the clue, without actually knowing which squares are black. This way, we can reduce the number of clauses in the CNF significantly, to 144 for such a clue.

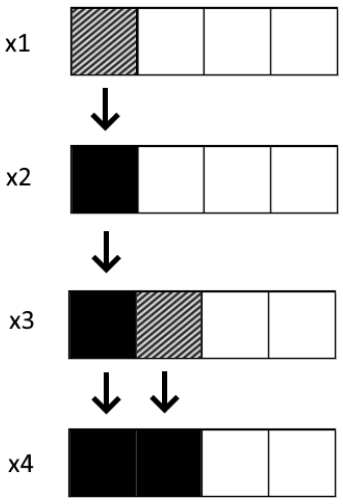
5.2 The Sequential Counter encoding

The Sequential Counter encoding is an encoding created by C. Sinz [18], which uses auxiliary variables to verify the *at-most-k* constraint in a sequence of variables, where at most k variables can be true.

The idea of the encoding is as follows: we have a sequence of variables $\{x_1, x_2, \ldots, x_n\}$, where each variable can be either true or false. We want to make sure that the number of true variables is at most k. To do this, we create a counter s_i for each variable, consisting of k booleans[1]. The segments of the counter will be named $s_{i,seg}$, where i is the number of the variable, and seg shows which segment in the counter the variable represents. For x_1, the counter would be $\{s_{1,1}, s_{1,2}, \ldots, s_{1,k}\}$. The goal of such a counter is to keep track of the number of variables that are true.

The Sequential Counter can be seen as follows; we will loop through the initial sequence of variables, and check whether the variable is true. If the variable is true, we will add 1 to the respective counter by setting the first segment that is false to true and copy this counter to the next variable's counter. If the variable is false, we only copy the current counter to the next variable's counter. This way, we can make sure that the last counter has the same number of true variables as the sequence.

Example 5.2.1. Suppose we have a sequence of variables $X_{\leq 4} = \{x_1, x_2, x_3, x_4\}$ where x_1 and x_3 are true, with the *at-most-4* constraint. We will loop through the variables in $X_{\leq 4}$, checking whether a variable is true. In this case, x_1 is true, so we set the first false segment in the respective counter to true and copy the counter to the counter of x_2. x_2 is false, so we only copy the segments from the counter of x_2 to x_3. x_3 is true, so we set the first false segment in the counter of x_3 to true, and copy all segments to the counter of x_4. x_4 is false, and since it is the last variable, we do not need to copy the counter to the next counter anymore.



[1]Though the paper by C. Sinz does not explicitly mention the number of segments a counter has, the encoding never uses variables in the counter after the k^{th} segment, even when there are n variables set to true. This is why we choose for the counters to have k segments.

The Sequential Counter encoding can be written as the following formula, which is a CNF:

$$\begin{aligned}
&(\neg x_1 \vee s_{1,1}) \wedge (\neg x_n \vee \neg s_{n-1,k}) \wedge \bigwedge_{1<j\leq k} (\neg s_{1,j}) \wedge \\
&\bigwedge_{1<i<n} \Big((\neg x_i \vee s_{i,1}) \wedge (\neg s_{i-1,1} \vee s_{i,1}) \wedge (\neg x_i \vee \neg s_{i-1,k}) \wedge \\
&\qquad \bigwedge_{1<j\leq k} \Big((\neg x_i \vee \neg s_{i-1,j-1} \vee s_{i,j}) \wedge (\neg s_{i-1,j} \vee s_{i,j}) \Big) \Big)
\end{aligned} \tag{5.1}$$

While this formula can easily be implemented in an encoding for a SAT solver, understanding it is more difficult. Therefore, we can rewrite every individual clause in the CNF with the rule:

$$\neg A \vee B \equiv A \rightarrow B \tag{5.2}$$

The arrow means that A implies B. What this essentially entails is that if A is true, B is also true. If A is false, B can be either true of false. For example, the first clause in the CNF states $(\neg x_1 \vee s_{1,1})$. This means that if x_1 is true, $s_{1,1}$ must also be true, effectively stating that counter s_1 is set to 1. If x_1 is false, $s_{1,1}$ can be either true or false. Rewriting every clause to this form gives us:

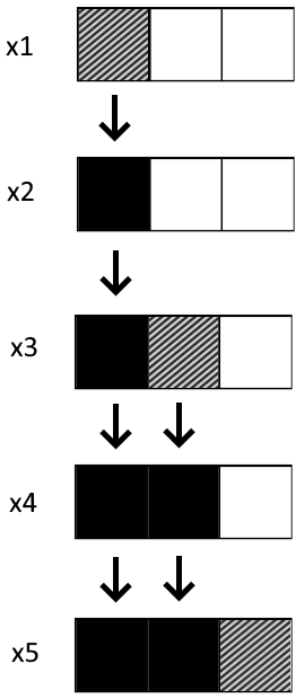
$$\begin{aligned}
&(x_1 \rightarrow s_{1,1}) \\
&(x_n \rightarrow \neg s_{n-1,k}) \\
&(\neg s_{1,j}) \qquad \text{for } 1 < j \leq k \\
&\left.\begin{aligned}
&(x_i \rightarrow s_{i,1}) \\
&(s_{i-1,1} \rightarrow s_{i,1}) \\
&(x_i \rightarrow \neg s_{i-1,k}) \\
&\left.\begin{aligned}
&((x_i \wedge s_{i-1,j-1}) \rightarrow s_{i,j}) \\
&(s_{i-1,j} \rightarrow s_{i,j})
\end{aligned}\right\} \text{for } 1 < j \leq k
\end{aligned}\right\} \text{for } 1 < i < n
\end{aligned} \tag{5.3}$$

Example 5.2.2. Suppose we want to check in a sequence of variables if *at-most-3* applies. We have a sequence of variables $X_{\leq 3} = \{x_1, x_2, x_3, x_4, x_5\}$, where the SAT solver evaluates $\{x_1, x_3, x_5\}$ to true. We create a counter for each variable, which is a sequence of 3 booleans, since $k = 3$:

$$\begin{aligned}
s_1 &= \{s_{1,1}, s_{1,2}, s_{1,3}\} \\
s_2 &= \{s_{2,1}, s_{2,2}, s_{2,3}\} \\
s_3 &= \{s_{3,1}, s_{3,2}, s_{3,3}\} \\
s_4 &= \{s_{4,1}, s_{4,2}, s_{4,3}\} \\
s_5 &= \{s_{5,1}, s_{5,2}, s_{5,3}\}
\end{aligned} \tag{5.4}$$

We can now convert this to a boolean formula with the clauses in either disjunction form or implicit form. Since implicit boolean formulas are more readable and understandable, we will use these. In table 5.6, a step-by-step of the boolean formula can be seen. Even though the SAT solver does not really evaluate the boolean formula this way, this is the most understandable way to show what exactly happens. Following is a small explanation of the steps in 5.6, together with a visual representation of the counters.

The SAT solver evaluated $\{x_1, x_3, x_5\}$ to true. Using this, line 1 states that $s_{1,1}$ needs to be evaluated to `true` as well, which can be seen in the counter of x_1. Segments $s_{1,2}$ and $s_{1,3}$ are evaluated to `false`. x_2 is evaluated to `false`, but since $s_{1,1}$ is evaluated to `true`, $s_{2,1}$ is as well (line 6), and the segment from the counter of x_1 is copied to the counter of x_2. Jumping forward to line 15, x_3 is evaluated to `true`, so $s_{3,1}$ is as well. Line 17 assures that the counter of x_3 did not already have 3 variables evaluated to `true`. In line 19, $s_{3,2}$ is evaluated to `true`, since the previous counter (x_2) had $s_{2,1}$ evaluated to true, and x_3 is also evaluated to `true`. This essentially increases the counter of x_3 by one. The same process happens for the counters of x_4 and x_5, where x_4 is evaluated to `false`, so the counter is the same as the counter of x_3, and x_5 is evaluated to `true`, so its respective counter gets increased by one. Lastly, on line 25, $s_{4,3}$ is evaluated to `false`, since x_5 is evaluated to `true`. This ensures that it is not the case that both the *at-most-3* constraint has been satisfied in the counter of x_4 and at the same time x_5 is also evaluated to `true`.



When trying to evaluate boolean formula 5.6, we end up with the following truth values for the counters:

$$\begin{aligned} s_1 &= \{s_{1,1}, \neg s_{1,2}, \neg s_{1,3}\} \\ s_2 &= \{s_{2,1}, \neg s_{2,2}, \neg s_{2,3}\} \\ s_3 &= \{s_{3,1}, s_{3,2}, \neg s_{3,3}\} \\ s_4 &= \{s_{4,1}, s_{4,2}, \neg s_{4,3}\} \\ s_5 &= \{s_{5,1}, s_{5,2}, s_{5,3}\} \end{aligned} \tag{5.5}$$

Since we can find a valid evaluation of the boolean formula, the variable sequence $X_{\leq 3}$ complies with the *at-most-3* constraint.

1	$(x_1 \to s_{1,1})$	x_1 is true, so $s_{1,1}$ must also be true
2	$(\neg s_{1,2})$	$s_{1,2}$ must be false
3	$(\neg s_{1,3})$	$s_{1,3}$ must be false
4	$i = 2$	
5	$(x_2 \to s_{2,1})$	x_2 is false, $s_{2,1}$ is unknown
6	$(s_{1,1} \to s_{2,1})$	$s_{1,1}$ is true, so $s_{2,1}$ must also be true (copying s_1 to s_2)
7	$(x_2 \to \neg s_{1,3})$	x_2 is false, $s_{1,3}$ is also false
8	$j = 2$	
9	$((x_2 \wedge s_{1,1}) \to s_{2,2})$	x_2 is false, so $s_{2,2}$ is unknown
10	$(s_{1,2} \to s_{2,2})$	$s_{1,2}$ is false
11	$j = 3$	
12	$((x_2 \wedge s_{1,3}) \to s_{2,3})$	x_2 is false, $s_{2,3}$ is unknown
13	$(s_{1,3} \to s_{2,3})$	$s_{1,3}$ is false
14	$i = 3$	
15	$(x_3 \to s_{3,1})$	x_3 is true, so $s_{3,1}$ must also be true
16	$(s_{2,1} \to s_{3,1})$	both are true
17	$(x_3 \to \neg s_{2,3})$	x_3 is true, so $s_{2,3}$ must be false (stating that the previous counter was not equal to 3)
18	$j = 2$	
19	$((x_3 \wedge s_{2,1}) \to s_{3,2})$	both are true, so $s_{3,2}$ is also true (s_3 now has 2 true values)
20	$(s_{2,2} \to s_{3,2})$	$s_{2,2}$ is false
21	$j = 3$	
22	$((x_3 \wedge s_{2,2}) \to s_{3,3})$	$s_{2,2}$ is false, so $s_{3,3}$ is unknown
23	$(s_{2,3} \to s_{3,3})$	$s_{2,3}$ is false
24	...	repeat for $i = 4$ and $i = 5$
25	$(x_5 \to \neg s_{4,3})$	x_5 is true, so $s_{4,3}$ cannot be true, otherwise there are more than 3 variables true in $X_{\leq 3}$

(5.6)

5.3 Extending the Sequential Counter encoding

The Sequential Counter encoding can be extended from an *at-most-k* constraint to an *exactly-k* constraint. In order to achieve this, we will add the following clauses to the CNF:

$$(x_1 \vee \neg s_{1,1}) \wedge (s_{n,k}) \wedge \bigwedge_{1<i<n} \Big((\neg s_{i,1} \vee x_i \vee s_{i-1,1}) \wedge \bigwedge_{1<j\leq k} \big((x_i \vee s_{i-1,j} \vee \neg s_{i,j}) \wedge (s_{i-1,j-1} \vee s_{i-1,j} \vee \neg s_{i,j}) \big) \Big) \tag{5.7}$$

These clauses check whether the last counter has the k^{th} segment set to true, which means that there are exactly k out of n variables true. Next to that, the clauses also ensure that if a segment is set to true, every segment in that counter before this segment is also true. This is needed because otherwise, the SAT solver would evaluate segment $s_{n,k}$ to true without necessarily evaluating the right number of variables to true. We can again rewrite the clauses to implicit form to make it easier to understand:

$$\left.\begin{array}{l} (s_{1,1} \rightarrow x_1) \\ (s_{n,k}) \\ \left.\begin{array}{l} (s_{i,1} \rightarrow (x_i \vee s_{i-1,1})) \\ (s_{i,j} \rightarrow (x_i \vee s_{i-1,j})) \\ (s_{i,j} \rightarrow (s_{i-1,j-1} \vee s_{i-1,j})) \end{array}\right\} \text{for } 1 < j \leq k \end{array}\right\} \text{for } 1 < i < n \tag{5.8}$$

5.4 Applying the Sequential Counter encoding

We can now apply the (extended) Sequential Counter encoding to our SAT encoding. We will use the Sequential Counter encoding for the *exactly-k* constraint. A cell with a clue is essentially the same problem as having exactly k out of n variables set to true. In this case, the clue in the cell is k, and a cell variable is set to true if it should be black in the solution.

For a cell containing a clue, we will create a counter for each of the surrounding cells, including the cell itself. The variables in the encoding will be the variables of the cells, denoted as $c_{x,y}$, where x and y are the coordinates of the cell. The segments of the counter will be changed to $s_{x,y,seg}$, where seg will still show which segment in the counter the variable represents. We will then encode this to a CNF with the formulas that are defined in sections 5.2 and 5.3. This will be done for every clue in the Mosaic configuration such that we end up with a CNF of the Mosaic, encoded using the Sequential Counter encoding. We can then evaluate this CNF formula with a SAT solver in the same way as described in chapter 3.3.

Example 5.4.1. Let's say we have the following configuration, depicted in figure 5.10. The cell in the middle ($c_{1,1}$) contains clue 5, meaning that there should be 5 black squares in the surrounding cells. We will start by creating counters for all cells:

$$\begin{aligned} s_{0,0} &= \{s_{0,0,1}, s_{0,0,2}, \ldots, s_{0,0,5}\} \\ s_{1,0} &= \{s_{1,0,1}, s_{1,0,2}, \ldots, s_{1,0,5}\} \\ &\ldots \\ s_{2,2} &= \{s_{2,2,1}, s_{2,2,2}, \ldots, s_{2,2,5}\} \end{aligned} \tag{5.9}$$

Now, we will encode the configuration of the Mosaic into a SAT encoding using the Extended Sequential Counter encoding as explained in chapter 5.3. Using this encoding, the SAT solver needs to evaluate the 5 segments of the last counter to `true` ($s_{2,2,1}, s_{2,2,2}, s_{2,2,3}, s_{2,2,4}, s_{2,2,5}$), meaning that exactly 5 out of the 9 cell variables have been evaluated to `true`.

When the SAT solver tries to evaluate 6 cell variables to true, like in figure 5.11, the SAT solver will evaluate the encoding as unsatisfiable, because of the clause $(\neg x_{2,2} \vee \neg s_{1,2,5})$. This clause states that the last cell variable can't be `true`, or the second-last counter can't have segment 5 being `true`. In this example, both are `true`, so this valuation is invalid.

A valid evaluation of the SAT solver can be seen in figure 5.12, where the SAT solver has evaluated 5 of the cell variables to `true`, and the 5 segments of the last counter have been evaluated to `true`.

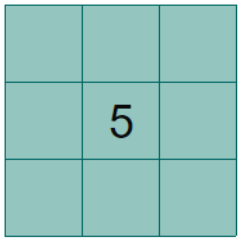


Figure 5.10: A simple configuration

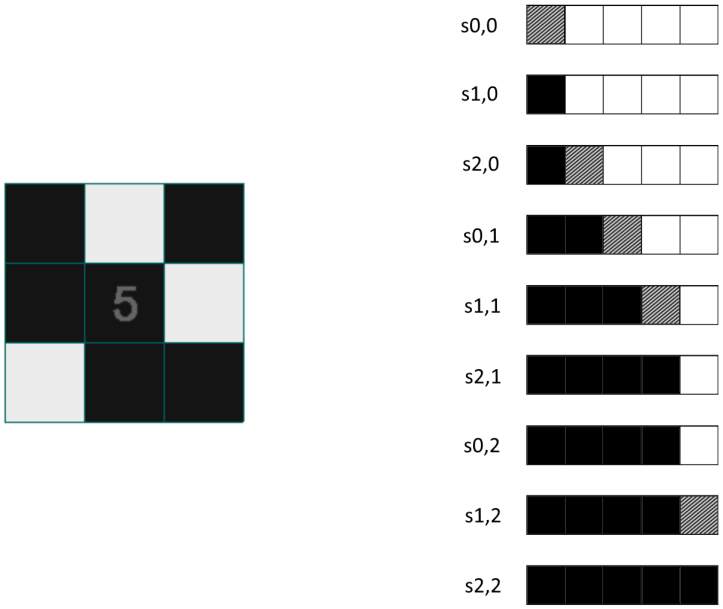


Figure 5.11: Invalid solution to the configuration with counters

Figure 5.12: Valid solution to the configuration with counters

Chapter 6

Experiments

In this chapter, we will look into both encodings and ways to optimize the encodings, as well as optimise the generation of puzzles. As explained in the previous chapters, there are two ways of encoding the CNF of a Mosaic puzzle; the naive encoding and the improved encoding. While the improved encoding uses fewer clauses and variables to encode the Mosaic puzzle, thus making solving a puzzle faster, it is also more complex.

We will look into solving and generating puzzles in different sizes and compare the time it takes, and the number of clauses they use. Next to that, we will try and generate puzzles with the naive encoding, and solve them with the improved encoding, and vice versa. We will also look at optimizing the improved encoding, by removing obsolete variables, and look if this makes any difference. Lastly, we will look at the way of generating puzzles. As explained in chapter 4, puzzles are now generated by checking every clue from left to right, top to bottom. A clear pattern can be seen here regarding clues that can be removed. We will try a different way of generating puzzles and will generate and solve the puzzles again with both methods.

6.1 Comparison of both encodings

To compare both encodings, we have downloaded 100 different puzzles from Simon Tatham's Portable Puzzle Collection [3], for the sizes $n * n$, where $n = \{5, 10, 15, 25, 50, 100\}$, giving a total of 600 puzzles for one encoding to solve. These puzzles are generated to be humanly solvable. We solved the downloaded puzzles with both encodings, and noted the total time it took to solve the puzzle. This includes both the creation of the encoding, as well as running the encoding with the SAT solver. After that, we took the mean per encoding per size, as well as the minimum time and the maximum time. The results can be seen in table 6.1 and figure 6.2.

Size	**Encoding**	**Mean (ms)**	**Minimum (ms)**	**Maximum (ms)**
5x5	Naive	5.94	2	32
5x5	Improved	3.44	2	11
10x10	Naive	14.51	9	25
10x10	Improved	5.58	4	7
15x15	Naive	35.46	26	55
15x15	Improved	10.34	8	14
25x25	Naive	117.57	90	158
25x25	Improved	26.85	22	31
50x50	Naive	781.3	637	904
50x50	Improved	182.01	155	225
100x100	Naive	3282.03	3051	4202
100x100	Improved	795.89	696	902

Table 6.1: Results of solving 100 human-solvable puzzles with both encodings

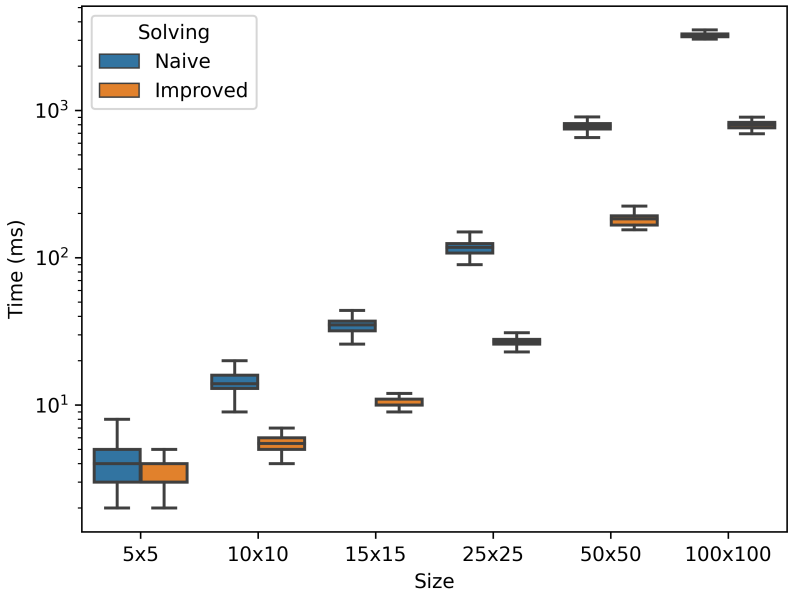


Figure 6.2: Results of solving 100 human-solvable puzzles with both encodings

Next, we compared the generation speed of the different encodings, and solved the generated puzzles with the other encoding. To test the limits of the generation methods, we gave both encodings 10 minutes to generate as many puzzles as possible. Both encodings did this for different sizes $n * n$, up to the point where both encodings could not generate a puzzle within 10 minutes anymore. After that, the other encoding solved all the puzzles. In order to make sure that the results are accurate, and that it is not the case that one poorly generated puzzle messes with the results, this process has been run ten times, and the average of the mean, min and max time have been taken. Next to that, the standard deviation is calculated over all data for that encoding and size. The results are listed in tables 6.3 and 6.4 and visualised in figures 6.5 and 6.6.

Here, we can again see that the improved encoding is significantly faster than the naive encoding. Where the improved encoding can generate around 10 puzzles of size 25x25 within 10 minutes, the naive encoding can't even generate a single puzzle. The naive encoding might be more straightforward to use and implement, but the improved encoding is significantly faster. While the difference is not huge for smaller puzzles, the naive encoding is no match for the improved encoding when the puzzles are even somewhat bigger.

In terms of solving the generated puzzles, not too much special is going on. The improved encoding still solves puzzles generated by the naive encoding faster than the naive encoding can solve puzzles generated by the improved encoding, as expected.

Size	**Encoding (generating)**	**No. of puzzles**	**Mean (ms)**	**Std. dev.**	**Min (ms)**	**Max (ms)**
5	Naive	5322.1	112.83	32.38	17.3	490.9
5	Improved	15055.9	39.84	9.73	8.8	254.0
10	Naive	90.1	6621.46	4537.69	1918.1	27875.3
10	Improved	800	753.62	187.12	393.5	1771.3
15	Naive	10.3	55032.31	31798.58	26176.6	119305.4
15	Improved	125.4	4793.54	1019.01	3052.9	8289.7
20	Naive	2.1	238230.58	78116.75	201728.9	272103.6
20	Improved	31.5	18931.08	3617.32	12712.5	26506.2
25	Naive	0.3	503974.67	85739.03	503974.67	503974.67
25	Improved	10.1	57382.01	11281.54	42959.1	73638.2

Table 6.3: Generating puzzles with both encodings, with a time limit of 10 minutes

Size	**Encoding (solving)**	**Mean (ms)**	**Std. dev.**	**Min (ms)**	**Max (ms)**
5	Improved	1.67	1.11	0.0	44.0
5	Naive	4.86	2.3	0.0	76.4
10	Improved	6.57	3.57	2.9	25.2
10	Naive	86.89	108.37	8.8	1178.7
15	Improved	20.65	10.45	11.3	39.6
15	Naive	359.5	295.76	52.7	1892.3
20	Improved	53.27	17.3	47.7	58.8
20	Naive	842.28	444.76	289.5	2100.9
25	Improved	115.67	65.43	115.67	115.67
25	Naive	1589.06	698.74	808.5	2950.0

Table 6.4: Solving the generated puzzles with the other encoding

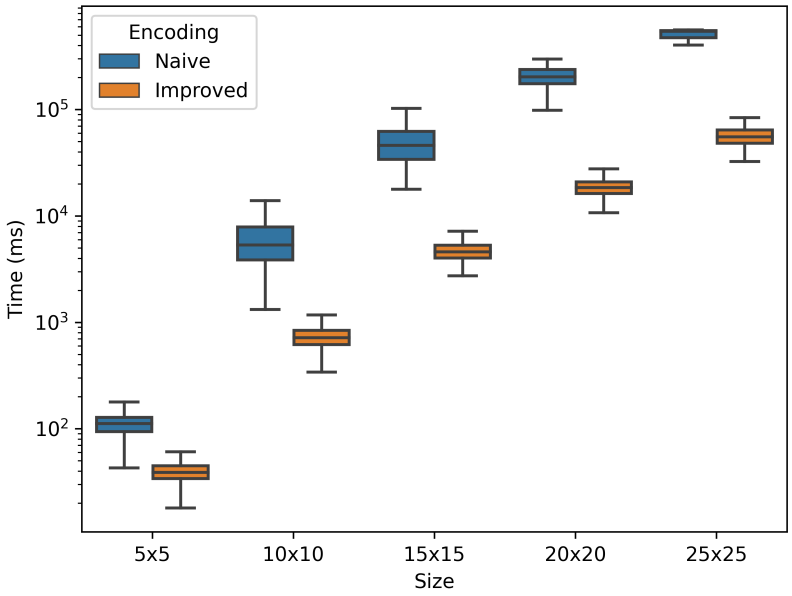


Figure 6.5: Generating puzzles with both encodings, with a time limit of 10 minutes

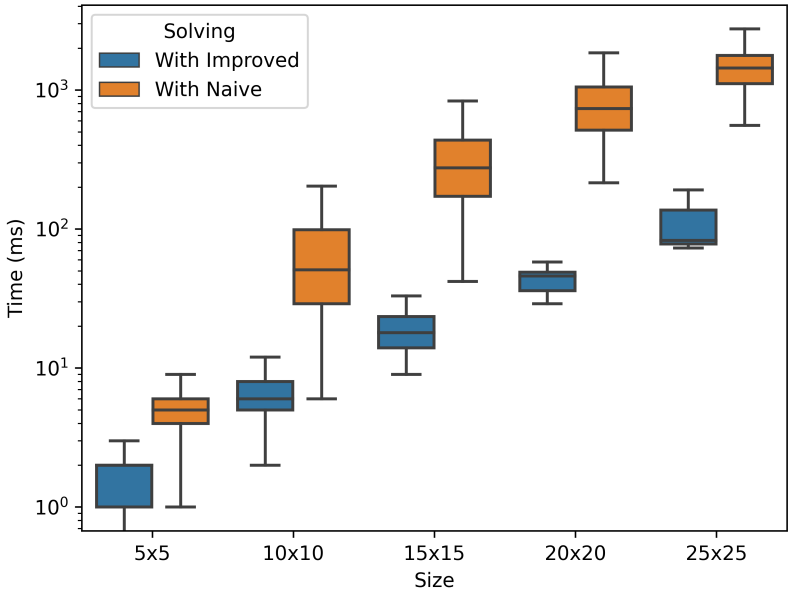


Figure 6.6: Solving the generated puzzles with the other encoding

6.2 Optimizing the improved encoding

Next, we try to optimize the improved encoding. Currently, in the improved method, counters are implemented for every cell. These counters can all count up to the value of the clue k, meaning they consist of k variables. This means that for one single clue in a 3x3 grid, there are $k * 9$ variables, plus the 9 variables of the cells.

While all cells do need a counter, not every cell necessarily needs a counter that can count up to k. In theory, the first cell's counter can only be 0, when there is a white square, or 1 when there is a black square. The counter of the second cell can only be 0, 1, or 2, depending on the first and second cells. If we continue this logic, every cell only needs a counter that goes up to the number of cell it is, or k. This way, only $(min(i, k)) * 9$ variables are needed for the counters in a 3x3 grid, instead of $k * 9$. As a result, there are also fewer clauses in this encoding. This because the improved encoding contains clauses which check segments of the first few counters that cannot possibly be evaluated to `true`. By removing those segments, we can also remove those clauses. Since there are fewer variables and clauses, we expected to see this back in the generation and solving speed.

To test this, we again solved the downloaded puzzles from Simon Tatham's Portable Puzzle Collection [3], for the sizes $n*n$, where $n = \{5, 10, 15, 25, 50, 100\}$. After that, we generated as many puzzles as possible within 10 minutes, for different sizes. We again ran this ten times to ensure that the results are not influenced by one poorly generated puzzle, and calculated the average of the mean, minimum and maximum times, together with the standard deviation over all data. The results can be found in tables 6.7 and 6.8, and are compared to the initial encodings in figures 6.9 and 6.10.

Even though it was first expected that the optimized version of the improved encoding would significantly speed things up, since it uses half as many variables as the original version of the improved encoding, this is not the case. The optimized encoding speeds up the process of solving the 100 puzzles in different sizes, albeit by a little.

When it comes to generating puzzles, the 'optimized' encoding is slightly faster to the original improved encoding for smaller puzzles, but as the puzzles get bigger, the 'optimized' encoding becomes slower than the original improved encoding, but not statistically significantly. The 'optimized' encoding can generate 105% of the total number of puzzles that the original improved encoding can generate at size 5. At size 25, the difference between the encodings is insignificant.

Even though the optimized encoding has fewer variables and clauses, the speed of solving and generating puzzles has not increased significantly. As explained above, the improved encoding contains clauses which check segments that cannot possibly be evaluated to `true`. Since we know that,

the clauses $\{(\neg s_{1,j})\ for\ 1 < j \leq k\}$ are added, which evaluate all segments except the first in the first counter to `false`. This way, by unit propagation, as explained in chapter 2.5, these segment variables are removed from the CNF. Hence, unit propagation likely causes the speed of the initial improved encoding come very close to the speed of the optimized encoding.

Size	**Mean (ms)**	**Minimum (ms)**	**Maximum (ms)**
5	3,51	2	51
10	5,57	4	71
15	9,14	7	13
25	23,34	20	28
50	155,84	135	200
100	714,92	598	801

Table 6.7: Solving 100 human-solvable puzzles with the optimized encoding

Size	**No. of puzzles**	**Mean (ms)**	**Std. dev.**	**Min (ms)**	**Max (ms)**
5	15896.7	37.75	8.69	11.1	256.0
10	823.9	731.38	195.89	393.0	1942.6
15	128.5	4680.55	1076.15	2741.8	8515.4
20	32	18493.64	3974.59	12509.0	27303.5
25	9.8	58403.89	14191.49	40875.2	79933.5

Table 6.8: Generating puzzles with the optimized encoding, with a time limit of 10 minutes

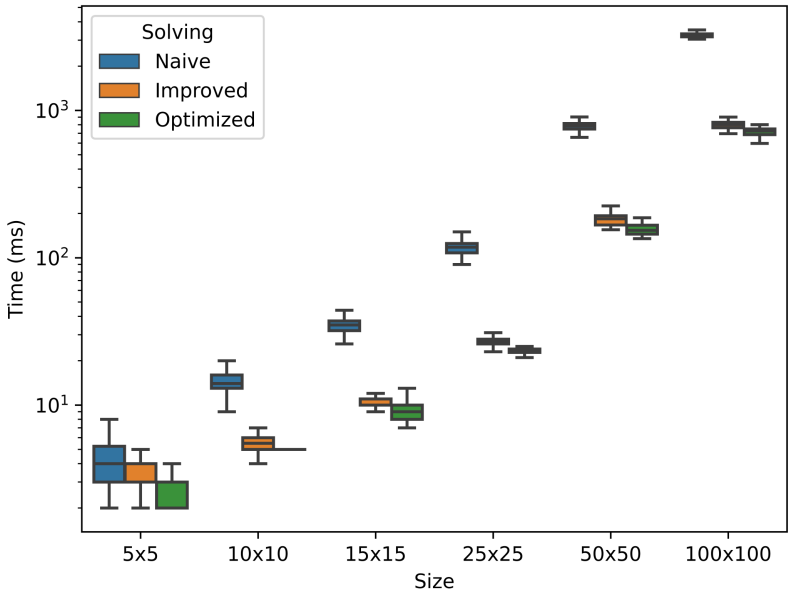


Figure 6.9: Solving 100 human-solvable puzzles with the optimized encoding, compared to the other encodings

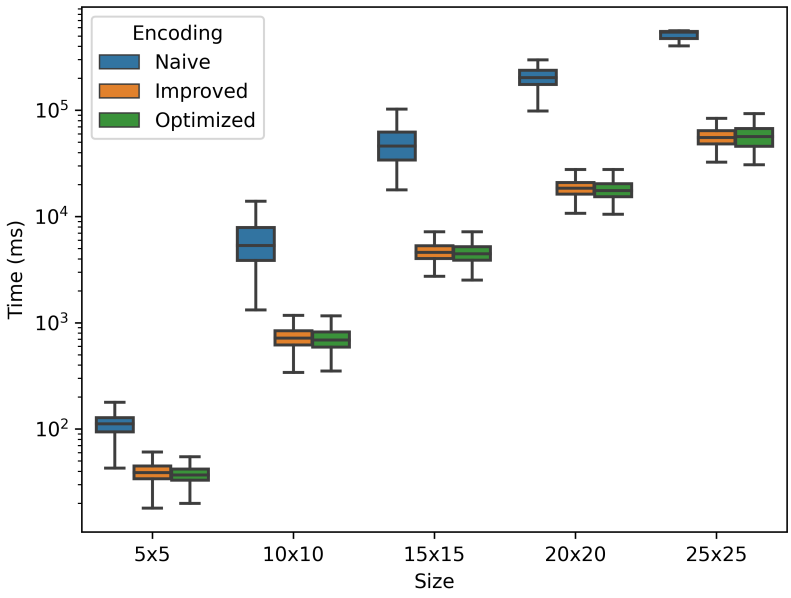


Figure 6.10: Generating puzzles with the optimized encoding with a time limit of 10 minutes, compared to the other encodings

6.3 A different approach to generating puzzles

Now that we looked into different encodings and optimization methods, we will lastly look into a different way of generating the puzzles. Currently, as explained in chapter 4, puzzles are generated by filling the entire grid with randomly chosen black and white squares, filling in the corresponding clue, and trying to remove clues one by one based on if this clue influences the uniqueness of the puzzle. The checking and removing of the clues is currently done from left to right, top to bottom. This way, a clear pattern can be seen regarding the generated puzzle. As can be seen in figure 6.11a, the top and left edges of the puzzle are empty, while further down and to the right, the number of clues becomes more dense.

Since the clues at the edges and in the corner have fewer neighbours to check, it is expected that having more clues on the edge and fewer clues in the middle speeds up the generation and solving process. A new approach to generating puzzles will be to randomize the process of checking and removing clues. This is done by creating an array of all possible coordinates and letting the generator pick a random coordinate pair from this array as clue to check. When the clue is checked, and either removed or put back in the puzzle, the coordinates will be removed from the array. This is done until the entire array is empty. As can be seen in figure 6.11b, there is a much more gradual division of the clues in the final generated puzzle.

With this new way of generating puzzles, we again generated and solved puzzles with both (original) encodings. The results of this can be seen in tables 6.12 and 6.13, and are compared to the original method in figures 6.14 and 6.15.

	1	2			6		4		4
	2			6	6	6		6	
					6	6	8		
	5		5						
	4	4			5	5		5	4
	4		2	3					
		0			0			4	2
							3	5	3
	2	0			2		2	4	2

(a) Left-right top-bottom generation

1			2			3		2	2
		2					4		
	2			6	6	6	7		4
					6	6	8		
5			5	7				6	
4		4		5			5	5	
4					3		4		3
		0			0				2
		0			2	3	3		
	2			2		3			2

(b) Random generation

Figure 6.11: Two differently generated puzzles, using the same starting configuration

The new approach to generating puzzles was chosen in order to make the clue distribution within the puzzles more even, since the first method of generating puzzles had little clues at the edges. The expectation was that generating puzzles using this method and solving those puzzles would be faster than the original method of generating puzzles, as clues around the edges have fewer neighbouring cells than clues in the middle of the puzzles.

As can be seen in figures 6.14 and 6.15, not only is generating puzzles with either encoding slower than the initial method, but the generated puzzles are also more difficult to solve for both encodings. A reason for this could be that the randomly generated puzzles have more clues than puzzles generated from the top-left to bottom-right, as almost all the clues at the edges can be removed while still maintaining a uniquely solvable puzzle. This means that there are more clues to encode in every step of the encoding phase of the puzzle to a CNF. Next to that, the resulting CNF after every step is bigger, because it has more clues, thus slowing the SAT solver down.

Size	**Encoding (generating)**	**No. of puzzles**	**Mean (ms)**	**Std. dev.**	**Min (ms)**	**Max (ms)**
5	Naive	5501.9	109.01	32.73	19.6	481.9
5	Improved	15434.5	38.85	9.94	8.7	242.9
10	Naive	62.4	9589.27	6414.43	2423.5	33220.1
10	Improved	638.6	941.22	281.96	394.5	2349.2
15	Naive	5	106890.41	49371.84	56356.9	178055.3
15	Improved	77.4	7760.28	2278.43	4109.0	14915.0
20	Naive	0.3	389355.33	160065.68	389355.33	389355.33
20	Improved	16.3	35760.11	10674.75	23144.9	61143.3
25	Naive	0	nan	nan	nan	nan
25	Improved	4.6	121047.4	26314.89	98810.7	146461.0

Table 6.12: Generating puzzles with the random generation

Size	Encoding (solving)	Mean (ms)	Std. dev.	Min (ms)	Max (ms)
5	Improved	1.64	1.07	0.0	41.7
5	Naive	4.65	2.03	0.0	67.1
10	Improved	9.45	5.28	3.8	31.0
10	Naive	139.33	149.32	10.1	1348.8
15	Improved	34.54	12.66	22.4	50.7
15	Naive	923.98	734.95	106.6	3842.7
20	Improved	123.67	103.65	123.67	123.67
20	Naive	2632.07	1709.27	947.7	6725.2
25	Improved	nan	nan	nan	nan
25	Naive	5794.16	2869.64	3428.1	9922.4

Table 6.13: Solving the randomly generated puzzles

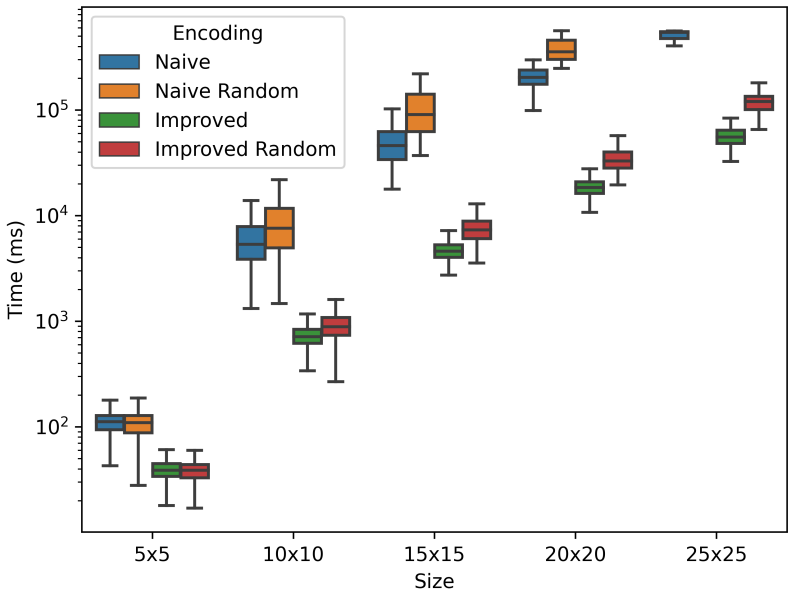


Figure 6.14: Generating puzzles with the random generation, compared to the initial method of generating puzzles for both encodings

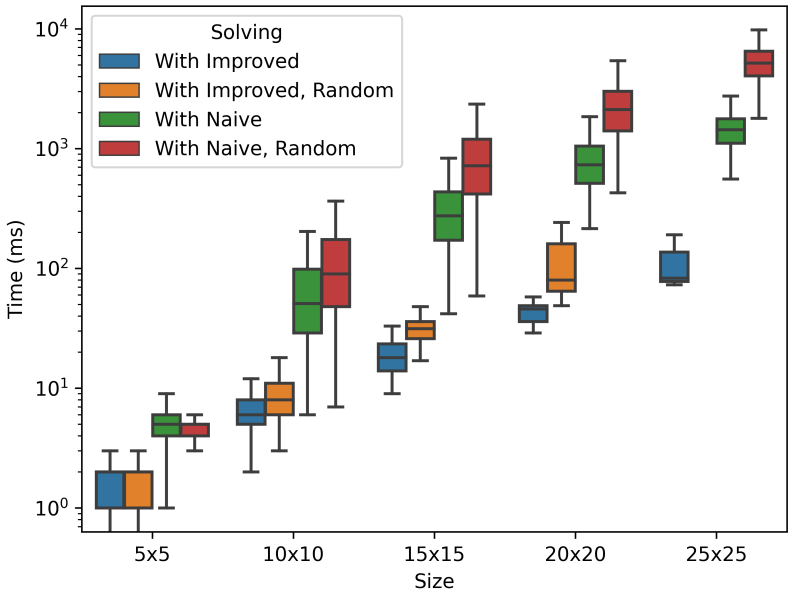
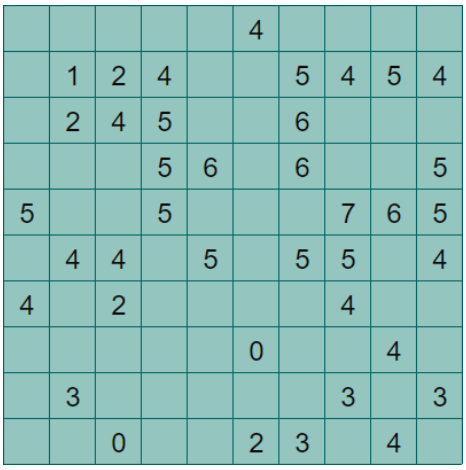
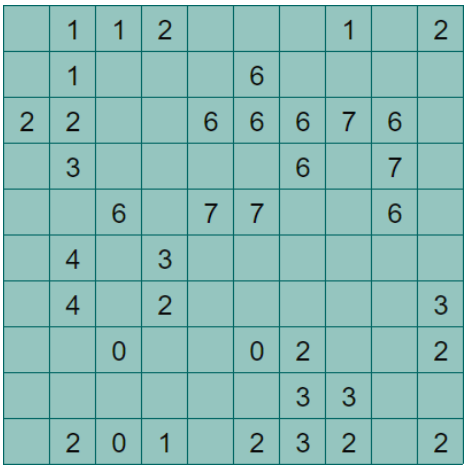


Figure 6.15: Solving the randomly generated puzzles, compared to solving the initially generated puzzles, with both methods

Even though the random generation method did not speed up the generation process, it did give new insights into possible generation methods. The generation process could be altered in a way that, instead of randomly selecting clues to check for removal, we could first check all 'easy' clues (0,1,8,9), such that the resulting configuration for the puzzle will be harder. The opposite can also be done; first checking all 'hard' clues (4,5,6), in order to make the puzzle easier. Next to that, instead of randomly determining the colour of the cells (black or white) we can select them based on an image, such that the solution of the puzzle will show a pixel-art like image. Examples of all three possible generation methods can be seen in figures 6.16 and 6.17.

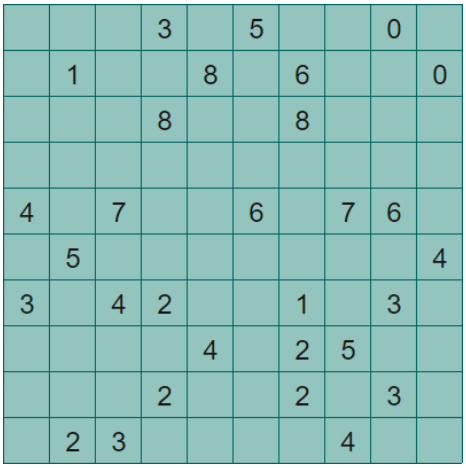


(a) Removing easy clues first to generate a harder puzzle



(b) Removing hard clues first to generate an easier puzzle

Figure 6.16: Generating different difficulties of puzzles



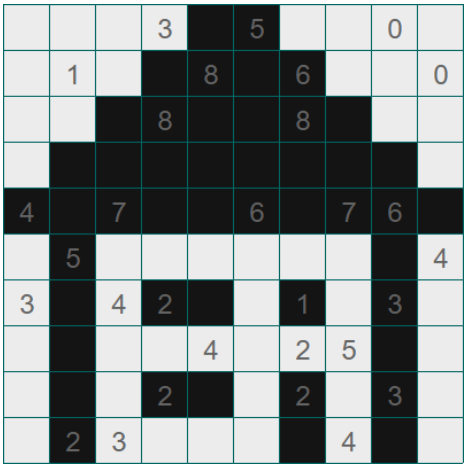


Figure 6.17: Puzzle generated with pixel-art as solution, together with the solution

For the two new generation methods, in which we can alter the difficulty of the generated puzzles, we again generated and solved puzzles with both (original) encodings, with a time limit of 10 minutes, where just as before, we ran this ten times to ensure that the results are not influenced by one poorly generated puzzle. We calculated the average of the mean, minimum and maximum times, together with the standard deviation over all data. The results can be seen in tables 6.18 and 6.19, and are visualised in figure 6.20.

Size	Encoding (generating)	No. of puzzles	Mean (ms)	Std. dev.	Min (ms)	Max (ms)
5	Naive Easy	7512.9	79.86	30.93	12.3	392.9
5	Naive Hard	4354.8	137.84	59.45	19.3	661.4
10	Naive Easy	135.3	4422.79	2963.56	1080.7	17965.6
10	Naive Hard	26.9	21848.72	13570.58	5962.6	59461.4
15	Naive Easy	13.6	42538.08	21053.07	17856.1	93300.0
15	Naive Hard	1.6	308155.15	87960.52	298146.7	318163.6
20	Naive Easy	2.6	197638.42	63902.42	156714.5	234063.8
20	Naive Hard	0.0	nan	nan	nan	nan
25	Naive Easy	0.7	473787.57	66808.5	473787.57	473787.57
25	Naive Hard	0.0	nan	nan	nan	nan

Table 6.18: Generating puzzles with the naive encoding, for an easy and a hard difficulty

Size	Encoding (generating)	No. of puzzles	Mean (ms)	Std. dev.	Min (ms)	Max (ms)
5	Improved Easy	19212.6	31.18	8.62	8.4	240.8
5	Improved Hard	15639.6	38.34	9.68	12.8	271.1
10	Improved Easy	911.5	658.85	181.69	310.0	1629.8
10	Improved Hard	522.6	1151.13	368.2	515.6	2998.8
15	Improved Easy	125.4	4786.58	1205.38	2610.4	8927.6
15	Improved Hard	51.6	11575.71	3561.29	6457.4	21690.2
20	Improved Easy	29.4	20211.47	4994.43	12618.3	33642.2
20	Improved Hard	9.0	63312.94	19572.93	39935.5	102090.0
25	Improved Easy	8.7	65783.13	12685.16	48038.2	83436.3
25	Improved Hard	2.3	235797.58	75853.75	213285.6	259053.6

Table 6.19: Generating puzzles with the improved encoding, for an easy and a hard difficulty

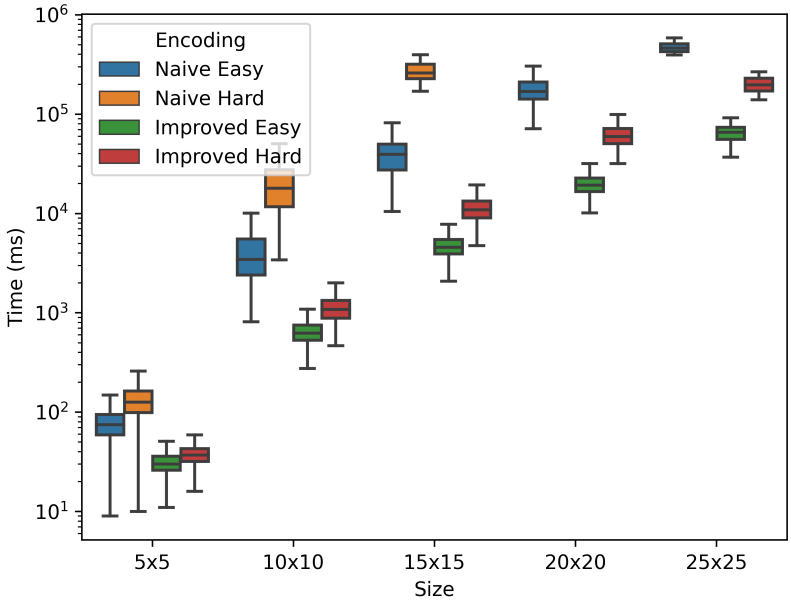


Figure 6.20: Generating puzzles with different difficulties, for both the naive and the improved encoding

From the tables and the boxplot it is already clear that for both the naive and the improved encoding, generating easy puzzles (where we first remove clues 4,5,6) is significantly faster than generating hard puzzles (where we first remove clues 0,1,8,9).

We also solved the generated puzzles with the other encoding, of which the results can be seen in tables 6.21 and 6.22, and are visualised in figure 6.23. Here, again it is very clear that puzzles generated in an easy way not only look easier, but are also easier for the SAT solver to solve. For a Mosaic puzzle of size 25x25, the naive encoding can solve a puzzle generated to be easy 20 times as fast on average as a puzzle generated to be hard. The most obvious explanation for this is that the easier puzzles contain more clues that require fewer clauses to solve these clues. So overall, encoding puzzles that are generated to be easy results in boolean formulas with fewer clauses, making the boolean formulas easier for the SAT solver as well.

Size	Encoding (solving)	Mean (ms)	Std. dev.	Min (ms)	Max (ms)
5	Improved Easy	0.96	0.92	0.0	40.5
5	Improved Hard	1.3	1.05	0.0	36.9
10	Improved Easy	6.14	3.36	2.2	23.7
10	Improved Hard	14.53	10.0	4.9	40.7
15	Improved Easy	23.3	14.36	11.3	55.8
15	Improved Hard	87.95	42.15	76.0	99.9
20	Improved Easy	63.98	23.64	49.1	79.5
20	Improved Hard	nan	nan	nan	nan
25	Improved Easy	124.43	26.84	124.43	124.43
25	Improved Hard	nan	nan	nan	nan

Table 6.21: Solving the naively generated puzzles with the improved encoding, for an easy and a hard difficulty

Size	Encoding (solving)	Mean (ms)	Std. dev.	Min (ms)	Max (ms)
5	Naive Easy	2.25	1.71	0.0	61.4
5	Naive Hard	4.74	3.74	0.0	79.0
10	Naive Easy	49.73	56.14	5.3	618.6
10	Naive Hard	335.56	363.84	18.7	3366.3
15	Naive Easy	231.42	186.1	29.7	1232.1
15	Naive Hard	2751.71	1990.73	443.0	9477.2
20	Naive Easy	628.45	453.24	147.9	2231.2
20	Naive Hard	9718.94	6147.92	3769.9	22254.7
25	Naive Easy	1267.36	645.5	542.2	2396.0
25	Naive Hard	24037.65	15983.97	18401.6	31295.5

Table 6.22: Solving the improved generated puzzles with the naive encoding, for an easy and a hard difficulty

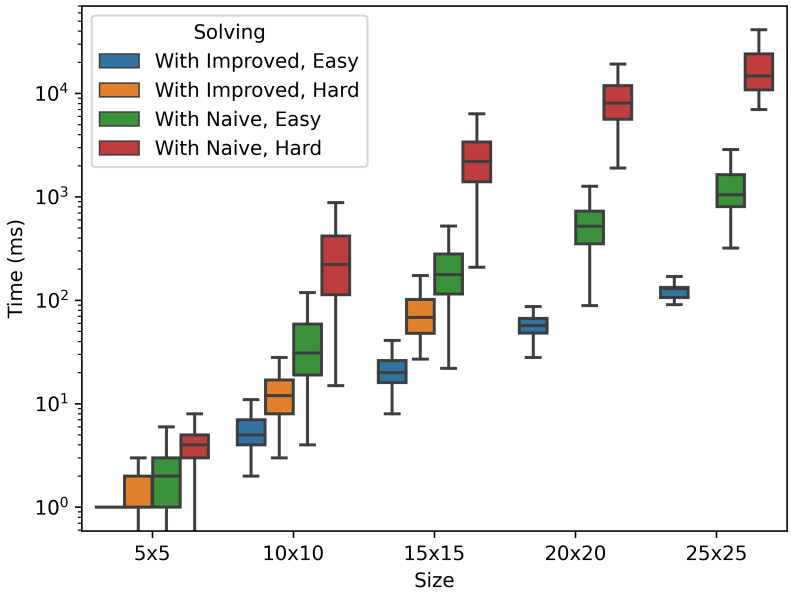


Figure 6.23: Solving the generated puzzles of different difficulties, for both the naive and the improved encoding

Chapter 7

Related Work

While Mosaic, also known as Fill-a-Pix, has already been approached as a logic puzzle to be solved with a SAT solver [16] and has been proven to be NP-Complete [17], no research can be found of finding ways to improve the already existing encodings of the Mosaic puzzle, as well as a way of generating Mosaic puzzles.

Since the invention of the Cook-Levin theorem, stating that the SAT problem is NP-Complete [7], a lot of research has been conducted on the SAT problem. In the JSAT Journal[1], much of this research is bundled. Next to that, there exist annual SAT Competitions[2], in which new SAT solvers are promoted and compared to already existing SAT solvers.

Logic puzzles are particularly popular concerning reducing problems to the SAT problem. The most famous logic puzzle, Sudoku, has been studied multiple times now [12] [15] [22]. Next to Sudoku, multiple other logic puzzles have been reduced to the SAT problem. Examples include Flood-It [21], Kamaji [9], the Binary puzzle [20] and Skyscrapers [11]. Mosaic has, just like Skyscrapers, extra difficulty in generating puzzles. Where with Sudoku, a solution can be generated and clues can be removed, with Mosaic and Skyscrapers, there are two layers of generation. This is because these puzzles do not only have a solution but also clues to come to this solution. In order to generate a puzzle, first a solution needs to be generated, after which all clues need to be calculated, while for Sudoku, only a solution needs to be generated in order to generate a puzzle. This makes generating puzzles more difficult.

[1] http://jsatjournal.org/
[2] http://www.satcompetition.org/

Chapter 8

Conclusions and Future Work

In this thesis, two different encodings for encoding the Mosaic puzzle as a SAT problem were introduced and tested against each other, for solving as well as generating puzzles. The experiments showed that the improved encoding is significantly faster than the original naive encoding, both with solving and generating the puzzles.

After testing both encodings, we tried to optimize the improved encoding by removing obsolete variables in the counters of the cells, but it turned out that this resulted in an encoding that was just as fast as the original improved encoding.

Lastly, we looked into a different way of generating the puzzles, by randomly removing clues instead of doing it in the same order every time. While the generated puzzles looked a lot more like the puzzles from Simon Tatham's Portable Puzzle Collection[3], it turned out that not only generating puzzles this way was slower, but the puzzles generated in this way were also more difficult to solve. From this, we can conclude that both optimizing the improved encoding as well as optimizing the generation method were not successful, though this did give us new insights for generating different difficulties of puzzles. when generating puzzles in an easy way, it does not only look easier, but it is also easier for the SAT solver to solve such puzzles.

Future research could be done to create a more reliable way of generating puzzles, for both humanly and non-humanly solvable puzzles. Experiments could also include letting humans try to solve the puzzles generated with the different generation methods.

Currently, the way of generating puzzles focuses on checking every cell once and returns one single uniquely solvable puzzle. Future research could be done to generate puzzles with the least amount of clues needed, or the number of clues needed on average for a certain size.

Bibliography

[1] Fill-a-pix. https://www.conceptispuzzles.com/index.aspx?uri=puzzle/fill-a-pix, 2022.

[2] Mosaic. https://www.puzzler.com/puzzles-a-z/mosaic, 2022.

[3] Mosaic, from simon tatham's portable puzzle collection. https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/mosaic.html, 2022.

[4] SAT competition 2009: Benchmark submission guidelines. http://www.satcompetition.org/2009/format-benchmarks2009.html, 2022.

[5] A. Biere, M. Heule, H. Van Maaren, and T. Walsh, editors. *Chapter 4: Conflict-driven clause learning SAT solvers*, pages 133–182. Frontiers in Artificial Intelligence and Applications. IOS Press BV, 2021.

[6] U. Bubeck and H. Büning. The power of auxiliary variables for propositional and quantified boolean formulas. *Studies in Logic*, 3:1–23, 2010.

[7] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[8] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2 edition, 2002.

[9] J. de Ruiter. On jigsaw sudoku puzzles and related topics. 2010.

[10] Y. Higuchi and K. Kimura. Np-completeness of fill-a-pix and $\sigma 2$ p-completeness of its fewest clues problem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E102A(11):1490–1496, 2019.

[11] L. Kolijn. Generating and solving skyscrapers puzzles using a SAT solver. 2022.

[12] G. Kwon and H. Jain. Optimized CNF encoding for sudoku puzzles. 01 2006.

[13] D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. 2-3.

[14] C.-M. Li and L. Anbulagan. Heuristics based on unit propagation for satisfiability problems. 1, 04 2000.

[15] I. Lynce and J. Ouaknine. Sudoku as a SAT problem. 01 2006.

[16] A. Myat, K. K. Htwe, and N. Funabiki. Fill-a-pix puzzle as a SAT problem. *2019 International Conference on Advanced Information Technologies (ICAIT)*, pages 244–249, 2019.

[17] J. Rosenhouse and J. Beineke. *The Mathematics of Various Entertaining Subjects: Research in Games, Graphs, Counting, and Complexity, Volume 2*. Princeton University Press, 09 2017.

[18] C. Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In P. van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[19] G. S. Tseitin. On the complexity of derivation in propositional calculus. 1983.

[20] P. Utomo and G. Pellikaan. Binary puzzle as a SAT problem. 2017.

[21] M. van Stiphout. Flood-it as a SAT problem. *bachelor thesis, Faculty of Science, Radboud University, Nijmegen, The Netherlands*, 2020.

[22] T. Weber. A SAT-based sudoku solver. In *LPAR*, pages 11–15, 2005.