Bachelor thesis
Computing Science

Radboud University

Generating and Solving Skyscrapers Puzzles Using a SAT Solver

Author:
Laura Kolijn
s1025724

First supervisor/assessor:
Dr. C.L.M. Kop
`c.kop@cs.ru.nl`

Second assessor:
Prof. dr. H. Zantema
`h.zantema@cs.ru.nl`

January 19, 2022

Abstract

The Skyscrapers puzzle is an NP-complete logical puzzle, meaning that a solution to this puzzle can be verified in polynomial time but as of yet there is no way to solve this puzzle in polynomial time. Additionally, every other problem that is in NP is reducible to the problem of solving the Skyscrapers puzzle. The aim of this thesis is to transform this puzzle into a Boolean satisfiability (SAT) problem and use a SAT solver to solve puzzles of any dimension. Using the SAT encoding and the SAT solver we will also create a program to generate new Skyscrapers puzzles. We experiment with a minimal and extended encoding for Latin Squares and find that the extended encoding provides a small advantage in time when generating puzzles with the Glucose SAT solver. With the Kissat SAT solver this effect is not evident. Furthermore, we experiment with two different ways of encoding the clues around the puzzle and we conclude that directly encoding the clues as variables provides a small advantage in time over encoding the consequences of the clues. Lastly, with the way the experiments are set up, the Glucose SAT solver seems to be more efficient when it comes to solving/generating of the Skyscrapers puzzle.

Contents

Chapter 1

Introduction

Logic puzzles, puzzles that can be solved using deduction techniques, have been gaining popularity over the years. The most famous logic puzzle probably being the Sudoku, there are many other kinds of logic puzzles of all shapes and sizes, like Slitherlink[1] and Akari[2]. The common denominator in most of these puzzles is that they are usually described using a concise and simple rule set, despite being very hard to solve. With the ever-rising increase in popularity also comes attention from the scientific world. Over the past years, a lot of research has been conducted on different logic puzzles and different aspects of these puzzles.

One of the lesser-known puzzles is the Skyscrapers puzzle (also known as Towers or the Building puzzle). This puzzle has been proven to be NP-complete [8]. A decision problem P is NP-complete if it satisfies two conditions. The first of the two conditions is that P needs to be in NP (nondeterministic polynomial time complexity class), which means that a solution to this problem can be verified in polynomial time but as of yet there is no way to solve the problem in polynomial time. The second condition is that every problem in NP is reducible to P in polynomial time.

In this thesis, the aim is to transform the Skyscrapers puzzle into another well-known NP-complete problem, namely a Boolean satisfiability problem. The resulting CNF formula for the Skyscrapers puzzle, can be used as input to a SAT solver to solve existing puzzles or to generate new puzzles. Showing how a logic puzzle can be solved as a SAT problem, can also be useful when solving other mathematical problems or even in real world applications.

First, chapter 2 provides some background information about the Skyscrapers puzzle, its rules, Conjunctive Normal Form and the Boolean satisfiability problem. In chapter 3, the translation from the Skyscrapers puzzle into a SAT problem will be described. Next, in chapter 4, some experiments conducted with this coding will be described. Chapter 5 will provide more information on other scientific research that has already been done on this subject. Finally, in chapter 6 the research is concluded.

[1]`https://www.nikoli.co.jp/en/puzzles/slitherlink/`

[2]`https://www.nikoli.co.jp/en/puzzles/akari/`

Chapter 2

Preliminaries

2.1 Skyscrapers

Skyscrapers, also called ‘Towers’ or ‘Building (City) Puzzle’ is a logical puzzle that was invented in 1992 by Masanori Natsuhara [4]. The Skyscrapers puzzle is a type of Latin square completion puzzle, like Sudoku. A Latin square is an n by n grid, filled with n different symbols. Each of these n symbols occurs exactly once in each row and exactly once in each column.

The Skyscrapers puzzle consists of an n by n grid with clues along the sides. The grid can be seen as the top view of a city(block) with skyscrapers of different heights in each cell.

The goal of the puzzle is to place a number between 1 and n, indicating the height of the skyscraper in each square in such a way that the grid forms a Latin square. Next to this, the clues along the sides of the grid indicate how many skyscrapers can be seen along that particular row or column when looking from that direction. A skyscraper can only be seen if there is no taller skyscraper in front of it blocking the view of the lower skyscraper, as can be seen in figure 2.1. Figure 2.2 shows an example of a Skyscrapers puzzle and its solution.

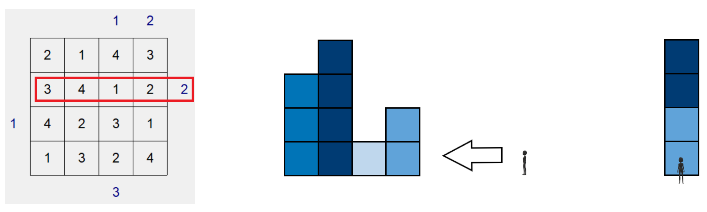
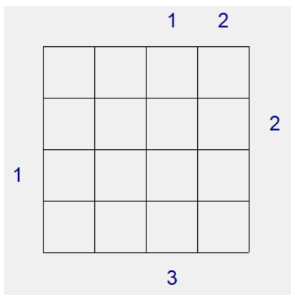
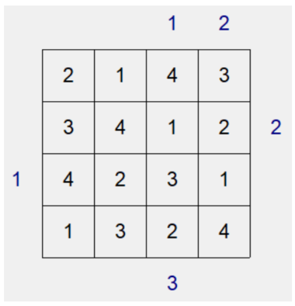


Figure 2.1: An example of how taller skyscrapers block the view of lower skyscrapers, with on the right the front view of what the person would see at the location of the clue



(a) An unsolved puzzle



(b) A solved puzzle

Figure 2.2: A Skyscrapers puzzle and its solution

2.2 Conjunctive Normal Form

In Boolean logic, a formula is in conjunctive normal form (CNF) if it is composed of a conjunction of clauses. Each of these clauses in turn consists of a disjunction of literals, a literal being a Boolean variable or its negation. Equation 2.1 would be an example of a formula in conjunctive normal form.

$$(a \lor b \lor d) \land (a \lor c) \land (\neg b \lor c) \land e \tag{2.1}$$

As can be sees from this equation, each clause in the conjunction consists either of a disjunction of literals or of a single literal. A clause that consists of a single literal is also called a unit clause, so in equation 2.1 "e" is a unit clause.

2.2.1 Rules of Replacement

In order to convert a formula in classical Boolean logic to a CNF formula, there are so-called rules of replacement that can be used. These rules of replacement are logical equivalences, of which the following equivalences are often used when converting a propositional formula into a CNF formula [16]:

- Double negation law:
$$\neg(\neg p) \equiv p \tag{2.2}$$
- Distributive law:
$$p \lor (q \land r) \equiv (p \lor q) \land (p \lor r) \tag{2.3}$$
- De Morgan's laws:
$$\neg(p \land q) \equiv \neg p \lor \neg q \tag{2.4}$$
$$\neg(p \lor q) \equiv \neg p \land \neg q \tag{2.5}$$

- Material implication:
$$p \to q \equiv \neg p \lor q \tag{2.6}$$

- Equivalence(s) involving bi-implications:
$$\begin{aligned} p \leftrightarrow q &\equiv (p \to q) \land (q \to p) \\ &\equiv (\neg p \lor q) \land (\neg q \lor p) \end{aligned} \tag{2.7}$$

2.2.2 DIMACS CNF

The DIMACS CNF format is a widely accepted standard format for CNF formulas. A file in this format is constructed as follows:

1. The first line(s) of the file may be comment lines. These lines start with a lowercase 'c' followed by a space and then the comment itself.

2. Next is the so-called "problem" line. This line starts with a lowercase 'p', followed by a space. Then the problem type is mentioned, which is 'cnf' for CNF files, followed by the number of variables in the encoding and the number of clauses in the encoding.

3. The next lines of the file consist of the clauses, with each of the clauses on a new line. Each clause is defined by listing the indices of the literals in the clause. For a positive literal, the index is used and for a negative literal, the negation of the index is used. These indices start at 1, since '0' marks the end of a clause.

The CNF file corresponding to (2.1) would look as follows:

```
c CNF file for the example formula
c
p cnf 5 4
1 2 4 0
1 3 0
-2 3 0
5 0
```

2.3 Boolean Satisfiability Problem

The Boolean satisfiability problem, abbreviated SATISFIABILITY, is the problem of determining whether there exists an assignment of values to the variables in a Boolean formula ϕ such that ϕ evaluates to `True`.

There are many variants of the SATISFIABILITY problem, one of them being CNF-SAT or SAT for short. CNF-SAT is the problem of determining the satisfiability of a Boolean formula in conjunctive normal form (CNF). For more information on formulas in CNF, refer back to section 2.2.

SAT was the first problem proven to be NP-complete. In 1971, Cook published his paper "The complexity of theorem-proving procedures" [6]. In this paper, Cook shows that any problem in the non-deterministic polynomial time complexity class (NP) can be reduced to an instance of the SAT-problem for CNF formulas, thereby proving that CNF-SAT is NP-complete.

2.4 SAT Solver

A SAT solver is a tool that takes a CNF formula f in DIMACS format (section 2.2.2) as input and outputs whether or not there exists an assignment to the variables that evaluates the formula to `True`. If there exists such an assignment, the SAT solver outputs 'SAT' (for "satisfiable") together with the satisfying assignment that was found. When no satisfying assignment exists, the SAT solver outputs 'UNSAT' (for "unsatisfiable").

There could be more than one assignment that satisfies the formula, but the only goal of the solver is to find whether or not there exists a satisfying assignment and therefore it will only output the assignment that was found.

The SAT solvers that will be used in the experiments in section 4 are the Glucose SAT solver [17] and the Kissat SAT Solver [3].

Glucose is a SAT solver that is heavily based on MiniSat [13], but it is more recently developed and, like most newer SAT solvers, it has the option to output a clausal proof for UNSAT. This clausal proof enables us to analyze exactly what clauses cause the solver to return 'UNSAT'.

Kissat is a new SAT solver, which won the two most recent SAT Competitions [2] of 2020 and 2021. This SAT solver is based on the CaDiCaL SAT solver, which is a lot less or even unaffected by the Minisat SAT solver. Therefore, this solver is ideally suited to use to make a comparison between two different SAT solvers.

Chapter 3

Translating Skyscrapers into a SAT problem

In order to be able to solve Skyscrapers with a SAT solver, the puzzle needs to be transformed into a CNF formula. In this chapter, it will be discussed how each Skyscrapers puzzle can be encoded as a SAT problem (section 2.3) such that the resulting CNF formula can be used as input to the SAT solver.

3.1 Constants

The first thing to establish are the aspects of the puzzle that cannot be influenced by the user: the constants. In the case of a Skyscrapers puzzle, there are three different constants.

- The first of these constants is a game parameter n, representing the dimension of the board. This parameter also dictates the maximum height a skyscraper in this puzzle can have, since each cell needs to contain a number between 1 and n.
- The second constant is a property of the field, namely the clues around the outside of the board. In order to access these clues, the following function is defined:

  `clues`(i, dir): Given a direction (North, East, South or West), this function produces the i^{th} clue that is located to the direction dir of the board. For any clues that are not filled in, 0 will be returned.

  For this function, $dir \in \{\text{North, East, South, West}\}$ and $i \in \{1, \ldots, n\}$.

- The third constant is the following function:

  `move`(dir): Given a cardinal direction (North, East, South or West), this function returns how the x coordinate and the y coordinate will change, respectively, when moving in that specified direction.
  For example, `move(North)` will return (0, -1) since the movement is to the cell above it in the same column. This means that the y coordinate does not change and 1 must be subtracted from the old x coordinate to arrive at the new x coordinate. Subtracting one when moving up may seem counter-intuitive, but that is because the top left cell in our grid has coordinates $(1, 1)$ as can be seen in figure 3.1.

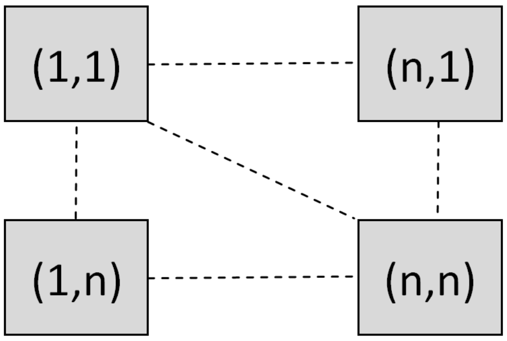


Figure 3.1: The coordinates of the cells in the puzzle.

3.2 Variables

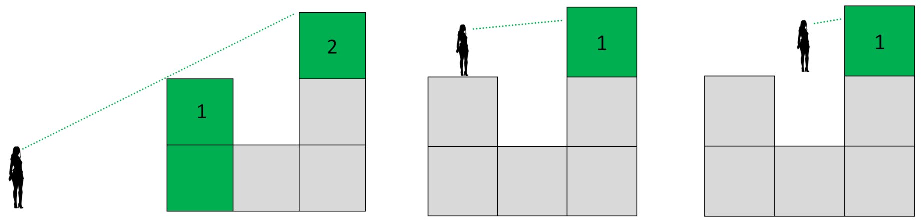
In addition to the constants defined in the previous section, two more variables are needed to translate Skyscrapers into a SAT problem.

The first variable to be used is the Boolean variable $s_{x,y,h}$. This variable indicates whether a skyscraper of height h is present in cell (x, y).
For this variable, $x, y, h \in \{1, \ldots, n\}$.

> **Example:**
> Assume we have a 3×3 grid.
> $s_{3,1,3}$ means that number 3 is in the last cell of the top row $(3, 1)$.
> $\neg s_{3,3,2}$ means that number 2 is *not* in the bottom right cell $(3, 3)$.

The next variable to be used is the Boolean variable $\texttt{visible}_{x,y,h,d,v}$. This variable is assigned true if and only if a person hovering above cell (x, y) at height h and looking in direction d, can see v skyscrapers that are taller than h. For this variable, $x, y \in \{0, \dots, n+1\}$, $h, v \in \{0, \dots, n\}$ and $d \in \{\text{North, East, South, West}\}$.

Example:
Assume a 3×3 grid with a clue 2 located to the left of the top row.
Figure 3.2a: A clue of 2 located to the left of the top row means that a person who is standing to the West of the top left cell $(x = 0, y = 1)$ on the ground $(h = 0)$, should be able to see a total of 2 skyscrapers from that point. This situation can be encoded as follows, using the `visible` variable: $\texttt{visible}_{0,1,0,\text{East},2}$.
Figure 3.2b: If a skyscraper of height 2 is in cell $(1, 1)$, this means that the person can see this skyscraper from their position West of the top left cell. This implies that the person, when hovering at height 2 above (or: standing on) this skyscraper in cell $(1, 1)$ and looking East, should be able to see one skyscraper with a height bigger than 2 from that point. This would give $\texttt{visible}_{1,1,2,\text{East},1}$.
Figure 3.2c: Subsequently, if a skyscraper of height 1 is in cell $(2, 1)$, this skyscraper is not visible for a person hovering at height 2 above cell $(1, 1)$ and looking East. This means that the person should still be able to see one skyscraper when they move to cell $(2, 1)$, hovering at height 2 and looking East. The resulting variable is then $\texttt{visible}_{2,1,2,\text{East},1}$.



(a) $\texttt{visible}_{0,1,0,\text{East},2}$ (b) $\texttt{visible}_{1,1,2,\text{East},1}$ (c) $\texttt{visible}_{2,1,2,\text{East},1}$

Figure 3.2: Example situations to illustrate the use of the `visible` variable. The (parts of the) skyscrapers that are visible, are colored green, with the numbers indicating the number of visible skyscrapers.

These variables are used together to encode the Skyscrapers puzzle into a SAT problem. In the end, the assignments to the Boolean variable $s_{x,y,h}$ will be used to represent the solution. The assignments to variable $\texttt{visible}_{x,y,h,d,v}$ on the other hand, are needed to represent the implications of the different clues around the board on the solution.

To arrive at a solution, it is necessary to ensure that the variables are subject to the constraints that are part of the ruleset for the Skyscrapers puzzle. Those constraints are defined in the next section.

3.3 Constraints

To implement the rules of the Skyscrapers puzzle described in section 2.1, several constraints must be added to the CNF encoding.

As can be read in section 2.1, the Skyscrapers puzzle is a type of Latin square completion puzzle. This implies that the constraints for a Latin square should be added to the encoding. There are two ways of encoding the Latin square completion problem: the "minimal encoding" and the "extended encoding" [7]. These ways of encoding are described in section 3.3.1 and section 3.3.2, respectively.

Next to this, there are the clues around the board that indicate how many skyscrapers are visible from that point of view. Section 3.3.3 describes the way of encoding these clues. Section 3.3.4 describes how to encode the different situations that can be derived from `visible` variables.

3.3.1 Latin Square Completion: Minimal Encoding

The "minimal encoding" for the Latin square completion problem states only the constraints that are absolutely necessary to represent the problem. For this encoding the following constraints need to be expressed:

1. There is at least one skyscraper (e.g., number) in each cell.

   More formally, this formula needs to express that each cell contains at least one of the numbers between 1 and n. This results in the following formula:

$$\bigwedge_{x=1}^{n} \bigwedge_{y=1}^{n} \bigvee_{h=1}^{n} s_{x,y,h} \tag{3.1}$$

2. A skyscraper of each height between 1 and n (e.g., every number between 1 and n) appears at most once in each row.

   More formally, this formula should express that in any row y the number h cannot be in both column x and column i where $x \neq i$ for any possible combination of x and i and for any value of the number h. This results in the following formula:

$$\bigwedge_{y=1}^{n} \bigwedge_{h=1}^{n} \bigwedge_{x=1}^{n} \bigwedge_{i=x+1}^{n} (\neg(s_{x,y,h} \wedge s_{i,y,h}))$$

   However, this formula is not CNF, meaning that the rules of replacement (section 2.2.1) must be used. In this case de Morgan's law (2.4) is applied, which gives the final CNF formula:

$$\bigwedge_{y=1}^{n} \bigwedge_{h=1}^{n} \bigwedge_{x=1}^{n} \bigwedge_{i=x+1}^{n} (\neg s_{x,y,h} \vee \neg s_{i,y,h}) \tag{3.2}$$

3. A skyscraper of each height between 1 and n (e.g., every number between 1 and n) appears at most once in each column.

   For this constraint, the same reasoning can be applied as for 2, but with rows and columns interchanged.

   This results in the following CNF formula for this constraint:

$$\bigwedge_{x=1}^{n} \bigwedge_{h=1}^{n} \bigwedge_{y=1}^{n} \bigwedge_{i=y+1}^{n} (\neg s_{x,y,h} \vee \neg s_{x,i,h}) \tag{3.3}$$

3.3.2 Latin Square Completion: Extended Encoding

As mentioned, it is also possible to encode the Latin square completion problem using the so-called "extended encoding". For this encoding, the constraints defined for the minimum encoding in section 3.3.1 are used, along with three additional constraints. These extra constraints are the following:

4. There is at most one skyscraper (e.g., number) in each cell.

   More formally, this formula needs to express that there cannot be a cell that contains two distinct numbers between 1 and n. This results in the following formula:

$$\bigwedge_{x=1}^{n} \bigwedge_{y=1}^{n} \bigwedge_{h=1}^{n+1} \bigwedge_{i=h+1}^{n} (\neg(s_{x,y,h} \wedge s_{x,y,i}))$$

   However, this formula is not CNF, meaning that the rules of replacement (section 2.2.1) must be used. De Morgan's law (2.4) can be applied in this case, which leads to the final CNF formula:

$$\bigwedge_{x=1}^{n} \bigwedge_{y=1}^{n} \bigwedge_{h=1}^{n+1} \bigwedge_{i=h+1}^{n} (\neg s_{x,y,h} \vee \neg s_{x,y,i}) \tag{3.4}$$

5. A skyscraper of each height between 1 and n (e.g., every number between 1 and n) appears at least once in each row.

   More formally, this formula needs to express that every number h (between 1 and n) should be contained in at least one of the columns x for every row y. This results in the following formula:

$$\bigwedge_{y=1}^{n} \bigwedge_{h=1}^{n} \bigvee_{x=1}^{n} s_{x,y,h} \tag{3.5}$$

6. A skyscraper of each height between 1 and n (e.g., every number between 1 and n) appears at least once in each column.

   Applying the same formal definition as for 5, but with rows replaced by columns and vice versa, yields the following formula:

$$\bigwedge_{x=1}^{n} \bigwedge_{h=1}^{n} \bigvee_{y=1}^{n} s_{x,y,h} \tag{3.6}$$

This extended encoding explicitly expresses the following properties of a Latin square:

- There is exactly one number in each cell.
- Each number between 1 and n appears exactly once in each row.
- Each number between 1 and n appears exactly once in each column.

However, the constraints in the minimal encoding already implicitly enforce these three properties.

While constraint 1 leaves room for more than one number in a cell, constraints 2 and 3 prevent this from actually happening. To illustrate this, take a row with a length of 3, where the first cell contains both numbers 1 and 2. This would mean that, according to constraint 2, the two remaining cells in the row could only contain the number 3, which in turn means that one cell will remain empty. However, constraint 1 does not allow for cells without a number, so it is not possible to have more than one number in a cell.

Similarly, constraints 2 and 3 leave room for a number not to appear in a row or column, since each number has to be at most once in each row and column. Having said that, if a number does not appear in a row or column, this would mean that one cell would remain empty and this is not allowed by constraint 1.

The extra clauses in the extended coding may seem redundant and therefore one would intuitively think that the SAT solver would take longer to solve the puzzle with the extended coding as opposed to the minimal coding. In some cases, however, the additional clauses help the SAT solver reach a conclusion more quickly regarding a possible solution to the problem at hand. This will be expanded upon in section 4.

3.3.3 Clues

In addition to the requirement that the resulting grid forms a Latin square, there are the clues around the board that must be taken into account. This section will describe two ways to encode these clues as a CNF formula: encoding the clues themselves or encoding the consequences of the clue for the first cell that is encountered from the position of the clue.

Encoding Clues

The first way of encoding the clues as a CNF formula is to express the clues as a `visible` variable, as was described in section 3.2. This results in the following CNF formula:

$$\bigwedge_{dir \in \{North, East, South, West\}} \bigwedge_{i=1}^{n} \text{Clues}_0(dir, i)$$

$$\text{Clues}_0(dir, i) = \begin{cases} \texttt{visible}_{i,0,0,\text{South},\texttt{clues}(i,dir)}, & \text{if } dir = \text{North and } \texttt{clues}(i, dir) \neq 0 \\ \texttt{visible}_{i,n+1,0,\text{North},\texttt{clues}(i,dir)}, & \text{if } dir = \text{South and } \texttt{clues}(i, dir) \neq 0 \\ \texttt{visible}_{0,i,0,\text{East},\texttt{clues}(i,dir)}, & \text{if } dir = \text{West and } \texttt{clues}(i, dir) \neq 0 \\ \texttt{visible}_{n+1,i,0,\text{West},\texttt{clues}(i,dir)}, & \text{if } dir = \text{East and } \texttt{clues}(i, dir) \neq 0 \end{cases} \tag{3.7}$$

Encoding Consequences of Clues

In addition to directly expressing the clues as `visible` variables, here too the clues can be rewritten in more elaborate encoding. To do this, the implications that the various clues have on the final solution must be expressed as a CNF formula. There are three different possibilities:

1. The clue is equal to 1.

   When this is the case, it is known that there can only be one skyscraper visible from the position of that clue. From this it follows that the first skyscraper that is encountered, needs to block the view of all skyscrapers behind it, which means that the first building that is seen from that clue needs to be height n. Since the first skyscraper blocks the view of all other skyscrapers in that particular row or column, it is not possible to reason further with this clue.

This results in the following CNF formula:

$$\bigwedge_{dir \in \{North, East, South, West\}} \bigwedge_{i=1}^{n} \text{Clues}_1(dir, i)$$

$$\text{Clues}_1(dir, i) = \begin{cases} s_{i,1,n}, & \text{if } dir = \text{North and } \texttt{clues}(i, dir) = 1 \\ s_{i,n,n}, & \text{if } dir = \text{South and } \texttt{clues}(i, dir) = 1 \\ s_{1,i,n}, & \text{if } dir = \text{West and } \texttt{clues}(i, dir) = 1 \\ s_{n,i,n}, & \text{if } dir = \text{East and } \texttt{clues}(i, dir) = 1 \end{cases} \tag{3.8}$$

2. The clue is equal to n.

   When this is the case, there should be n skyscrapers visible from the position of that clue. Since there are exactly n cells in each row and column, this means that no skyscraper can be blocking the view of another skyscraper. From this it follows that the skyscrapers have to be arranged in ascending order seen from the position of that clue.

   This results in the following CNF formula:

$$\bigwedge_{dir \in \{North, East, South, West\}} \bigwedge_{i=1}^{n} \bigwedge_{k=1}^{n} \text{Clues}_2(dir, i, k)$$

$$\text{Clues}_2(dir, i, k) = \begin{cases} s_{i,k,k}, & \text{if } dir = \text{North and } \texttt{clues}(i, dir) = n \\ s_{i,(n-k+1),k}, & \text{if } dir = \text{South and } \texttt{clues}(i, dir) = n \\ s_{k,i,k}, & \text{if } dir = \text{West and } \texttt{clues}(i, dir) = n \\ s_{(n-k+1),i,k}, & \text{if } dir = \text{East and } \texttt{clues}(i, dir) = n \end{cases} \tag{3.9}$$

3. The clue is equal to any number between 1 and n.

   When this is the case, it is not possible for any cell in the row or column associated with the clue to directly infer which number it should contain. However, some numbers can be excluded for the first cell(s) in the concerned row or column.

Example:
Suppose there is a board of 4×4 which contains a row with clue 3. Then for the first cell in that row, the values 4 and 3 can be excluded. This is because a 4 in that cell would mean that the view of all skyscrapers is blocked, so that only 1 skyscraper is visible instead of 3. Similarly, a 3 in that cell would mean that only a skyscraper with height 4 further down the row could still be visible making a total of 2 skyscrapers visible instead of the required 3.
For the second cell, the value 4 can be excluded: a skyscraper with the maximum height in this cell would mean that only this skyscraper and the one before it are visible, resulting in a total of 2 visible skyscrapers instead of 3.

Given a clue c, it is known that there must be c skyscrapers visible from the position of that clue. A skyscraper of height h in the first cell seen from the clue, means that skyscrapers further in the row or column with a height between 1 and $h-1$ will not be visible. Given that c skyscrapers should be visible, the first skyscraper can block the view of at most $n-c$ skyscrapers. This gives $h-1 \leq n-c$, which can be rewritten to $h \leq n-c+1$, meaning that the height of the first skyscraper cannot exceed $n-c+1$.
This idea can be generalized to apply it to all the cell(s): for the i^{th} cell in a row or column with clue c, the value cannot be any of the numbers starting from $n-c+i+1$ up to and including n.

This results in the following CNF formula:

$$\bigwedge_{\substack{dir \in \{\text{North, East,} \\ \text{South, West}\}}} \bigwedge_{i=1}^{n} \bigwedge_{k=1}^{(\texttt{clues}(i,dir)-1)} \bigwedge_{o=n-\texttt{clues}(i,dir)+1+k}^{n} \text{Clues}_3(dir,i,k,o)$$

$$\text{Clues}_3(dir,i,k,o) = \begin{cases} \neg s_{i,k,o}, & \text{if } dir = \text{North and} \\ & 1 < \texttt{clues}(i,dir) < n \\ \neg s_{i,(n-k+1),o}, & \text{if } dir = \text{South and} \\ & 1 < \texttt{clues}(i,dir) < n \\ \neg s_{k,i,o}, & \text{if } dir = \text{West and} \\ & 1 < \texttt{clues}(i,dir) < n \\ \neg s_{(n-k+1),i,o}, & \text{if } dir = \text{East and} \\ & 1 < \texttt{clues}(i,dir) < n \end{cases} \tag{3.10}$$

In addition to excluding some values for certain cells, something can also be said about the effect of placing any of the values not excluded by (3.10) in the first cell next to the clue. If a skyscraper of any height is placed in the first cell next to a clue, it will always be visible, regardless of its height. This means that it needs to be expressed that the number of skyscrapers that must be visible from that first cell is one less than the number indicated by the value of the clue. In other words, if a skyscraper of a certain height is placed in the first cell, then a person hovering directly above that skyscraper should be able to see exactly clue minus one skyscraper higher than the first skyscraper when looking directly ahead. To express this, the `visible` variable introduced in section 3.2 is used.

This results in the following CNF formula:

$$\bigwedge_{dir \in \{North, East, South, West\}} \bigwedge_{i=1}^{n} \bigwedge_{k=1}^{n-\texttt{clues}(i,dir)+1} \text{Clues}_4(dir, i, k)$$

with

$$\text{Clues}_4(dir,i,k,o) = \begin{cases} s_{i,1,k} \rightarrow \texttt{visible}_{i,1,k,\text{South},\texttt{clues}(i,dir)-1}, & \text{if } dir = \text{North and} \\ & 1 < \texttt{clues}(i,dir) < n \\ s_{i,n,k} \rightarrow \texttt{visible}_{i,n,k,\text{North},\texttt{clues}(i,dir)-1}, & \text{if } dir = \text{South and} \\ & 1 < \texttt{clues}(i,dir) < n \\ s_{1,i,k} \rightarrow \texttt{visible}_{1,i,k,\text{East},\texttt{clues}(i,dir)-1}, & \text{if } dir = \text{West and} \\ & 1 < \texttt{clues}(i,dir) < n \\ s_{n,i,k} \rightarrow \texttt{visible}_{n,i,k,\text{West},\texttt{clues}(i,dir)-1}, & \text{if } dir = \text{East and} \\ & 1 < \texttt{clues}(i,dir) < n \end{cases}$$

$$\equiv \begin{cases} \neg s_{i,1,k} \vee \texttt{visible}_{i,1,k,\text{South},\texttt{clues}(i,dir)-1}, & \text{if } dir = \text{North and} \\ & 1 < \texttt{clues}(i,dir) < n \\ \neg s_{i,n,k} \vee \texttt{visible}_{i,n,k,\text{North},\texttt{clues}(i,dir)-1}, & \text{if } dir = \text{South and} \\ & 1 < \texttt{clues}(i,dir) < n \\ \neg s_{1,i,k} \vee \texttt{visible}_{1,i,k,\text{East},\texttt{clues}(i,dir)-1}, & \text{if } dir = \text{West and} \\ & 1 < \texttt{clues}(i,dir) < n \\ \neg s_{n,i,k} \vee \texttt{visible}_{n,i,k,\text{West},\texttt{clues}(i,dir)-1}, & \text{if } dir = \text{East and} \\ & 1 < \texttt{clues}(i,dir) < n \end{cases} \tag{3.11}$$

3.3.4 Reasoning With the `visible` Variable

Defining all possible `visible` variables would result in a huge number of variables, most of which are not needed to solve the problem. Therefore, only those `visible` variables will be used that are required to encode the clues and that are derived from further reasoning with these variables, as can be seen in algorithm 1. There are three different cases to consider when reasoning with `visible` variables:

(i) The variable indicates that you are in the last cell (looking from the clue), but either there should still be skyscrapers visible or you have not yet encountered the highest building. Being in the last cell is indicated by either an x or a y coordinate with a value less than one or greater than the dimensions of the board when moving one cell further in the direction corresponding to the clue.

(ii) The variable indicates that you should be able to see x number of skyscrapers, but you are less than x cells away from the last cell.

(iii) The cases that are not covered by (i) or (ii).

Cases (i) and (ii) have the same outcome with regards to the clauses that will be added to the CNF formula. The variable associated with either of these situations will evaluate to false since both cases are impossible:

- Case (i) is impossible, because in the last cell there cannot be any skyscrapers left to see if there is no place where that skyscraper can be.
- Case (ii) is impossible for a similar reason, namely that no x number of skyscrapers can be seen if there are not enough cells left to place this number of skyscrapers.

Suppose the variable being reasoned with is $\texttt{visible}_{x,y,h,d,v}$, then the following unit clause is added to the CNF formula:

$$\neg\texttt{visible}_{x,y,h,d,v} \tag{3.12}$$

For case (iii), every value between 1 and n for the next cell is considered and the consequences of that particular value on, for example, how many skyscrapers are visible are looked at. There are three situations that need to be looked at:

1. In case the current `visible` variable indicates that no skyscrapers can be seen straight ahead when hovering at height h, there cannot be a skyscraper with a height greater than h in the next cell. If there would be a skyscraper ahead with a height greater than h, this skyscraper would have been visible.
2. If the skyscraper in the next cell has a greater height than the height being hovered at according to the current `visible` variable, for the next cell the height being hovered at increases to that greater value. If a skyscraper has a height greater than the height being hovered at, it will be visible when looking directly ahead and therefore the number of skyscrapers visible from the next cell will be one less than the number visible from the current cell.
3. In case the skyscraper in the next cell has a lesser height than the height being hovered at according to the current `visible` variable, then the height being hovered at will not change. Furthermore, the same number of skyscrapers will be visible from the next cell, since a skyscraper with a height less than the height being hovered at will not be visible when looking directly ahead.

Suppose that the variable being reasoned with is $\texttt{visible}_{x,y,h,d,v}$, then this results in the following clauses, where $(d_0, d_1) = \texttt{move}(d)$:

$$\begin{aligned}
&\bigwedge_{i=1}^{n} \begin{cases} \texttt{visible}_{x,y,h,d,v} \to \neg s_{x+d_0,y+d_1,i}, & \text{if } v = 0 \text{ and } i > h \\ \texttt{visible}_{x,y,h,d,v} \to \left(s_{x+d_0,y+d_1,i} \to \texttt{visible}_{x+d_0,y+d_1,i,d,v-1}\right), & \text{if } i > h \\ \texttt{visible}_{x,y,h,d,v} \to \left(s_{x+d_0,y+d_1,i} \to \texttt{visible}_{x+d_0,y+d_1,h,d,v}\right), & \text{otherwise} \end{cases} \\
\equiv &\bigwedge_{i=1}^{n} \begin{cases} \neg\texttt{visible}_{x,y,h,d,v} \vee \neg s_{x+d_0,y+d_1,i}, & \text{if } v = 0 \text{ and } i > h \\ \neg\texttt{visible}_{x,y,h,d,v} \vee \left(s_{x+d_0,y+d_1,i} \to \texttt{visible}_{x+d_0,y+d_1,i,d,v-1}\right), & \text{if } i > h \\ \neg\texttt{visible}_{x,y,h,d,v} \vee \left(s_{x+d_0,y+d_1,i} \to \texttt{visible}_{x+d_0,y+d_1,h,d,v}\right), & \text{otherwise} \end{cases} \\
\equiv &\bigwedge_{i=1}^{n} \begin{cases} \neg\texttt{visible}_{x,y,h,d,v} \vee \neg s_{x+d_0,y+d_1,i}, & \text{if } v = 0 \text{ and } i > h \\ \neg\texttt{visible}_{x,y,h,d,v} \vee \neg s_{x+d_0,y+d_1,i} \vee \texttt{visible}_{x+d_0,y+d_1,i,d,v-1}, & \text{if } i > h \\ \neg\texttt{visible}_{x,y,h,d,v} \vee \neg s_{x+d_0,y+d_1,i} \vee \texttt{visible}_{x+d_0,y+d_1,h,d,v}, & \text{otherwise} \end{cases}
\end{aligned} \tag{3.13}$$

Looking back at figure 3.2b, the third situation would apply: the variable $\texttt{visible}_{1,1,2,\mathrm{East},1}$ indicates that one skyscraper should be visible, so the first situation does not apply. In addition, the skyscraper at the position adjacent to the person's current position is lower than the height at which the person is hovering so the second situation is dropped. This would leave us with the following formula: $\neg\texttt{visible}_{1,1,2,\mathrm{East},1} \vee \neg s_{2,1,1} \vee \texttt{visible}_{2,1,2,\mathrm{East},1}$.

In figure 3.2c, the second situation would apply since there still needs to be one skyscraper visible according to the variable $\texttt{visible}_{2,1,2,\mathrm{East},1}$ and in addition to that the skyscraper that the person is looking at is higher than the current height that the person is hovering at. This would result in the following clause being added: $\neg\texttt{visible}_{2,1,2,\mathrm{East},1} \vee \neg s_{3,1,3} \vee \texttt{visible}_{3,1,3,\mathrm{East},0}$.

What is described in this section can be represented as an algorithm as follows:

Algorithm 1 Reasoning with `visible` variables

```
Input: A list $L$ of all visible variables encountered when applying (3.11).
  for Every variable $v$ with parameters $x, y, h, d, v$ in $L$ do
    Apply (3.12) or (3.13) based on the parameters of $v$.
    Add new visible variables that are encountered to $L$.
  end for
```

3.4 Generating Puzzles

With the CNF encoding that is created using the set of constraints that were defined in section 3.3, any Skyscrapers puzzle can be solved. This was confirmed by solving ten different puzzles downloaded from Simon Tatham's Portable Puzzle Collection [18] and checking whether the solution that the SAT solver came up with, matched the solution given on the website.

Next to solving Skyscrapers puzzles, the CNF encoding can also be used to generate new puzzles. To generate new puzzles, a similar strategy will be used as is described in [5] for Sudoku puzzles. To start with, a random solution to a Skyscrapers puzzle will be generated. This is done by first calling the SAT solver with either the minimal or extended encoding to create a Latin square. To ensure that the SAT solver will create a unique Latin square every time, a random seed is passed to the SAT solver. With the Latin square that the SAT solver outputs, the clues that are associated with this Latin square can be computed.

To make sure that the set of clues that is computed, yields a unique solution to this puzzle, the SAT solver is run again. This time, the constraints for the clues are added to the CNF encoding and on top of that an additional constraint that blocks the solution that was generated earlier is added. This additional constraint looks as follows:

$$\bigvee_{z=G_{x,y}} \neg s_{x,y,z}, \text{ where } G_{x,y} \text{ is the value of cell } (x, y) \text{ in the solution} \tag{3.14}$$

If the puzzle has a unique solution, the SAT solver will return 'UNSAT', since the only solution to the puzzle was blocked in the CNF encoding. If the puzzle does not have a unique solution, the SAT solver will return another solution than the one that was randomly generated. When this is the case, the cells that have different values in the two solutions will be identified. One of these cells is then chosen at random and the value that was in that cell in the original solution is fixed. After adding this fixed value to the encoding, the SAT solver is run again to see whether the puzzle configuration is unique. If fixing the value also does not produce a unique puzzle configuration, the process is repeated: the original field that was generated is compared to the new solution given by the SAT solver and one value that differs between the two fields is fixed. This process of fixing one value at a time is repeated until the puzzle configuration is unique.

The focus in generating the puzzles is not on finding a puzzle configuration with as few clues as possible, but on finding any unique puzzle configuration. Thus, all puzzles that are generated have $4 \times n$ clues around the board (where n is the dimension of the board) and zero or more values that are fixed on the board.

Chapter 4

Experiments

4.1 Comparison of Encodings and SAT Solvers

As can be read in section 3.3, there are two different ways of encoding possible for the Latin square constraint: the minimal encoding and the extended encoding. Despite the fact that the time to generate the encoding will increase due to the additional clauses added for the extended encoding, it may be the case that the total time to generate puzzles decreases. The reason for this is that the additional clauses allow the SAT solver to apply more optimizations, for example. This experiment will look at the difference in time taken to generate new puzzles of different sizes when the minimal encoding is used and when the extended encoding is used.

In addition, as can be read in section 3.3.3, there are two different ways to encode the clues around the puzzle. In this experiment the difference in efficiency between the two ways of encoding the clues is being looked at.

Next to this, there are also a lot of different SAT solvers. Each SAT solver uses a slightly different technique to solve the Boolean satisfiability problem it received as input. This can lead to differences in the time it takes to solve a particular problem. This experiment uses two SAT solvers, described in section 2.4, and compares the time required to generate new puzzles of different sizes. The code for these experiments can be found at `https://gitlab.science.ru.nl/lkolijn/Skyscrapers`.

4.1.1 Set Up of the Experiments

A script is run to generate one hundred puzzles with the consequences of the clues encoded (section 3.3.3), both with the minimal and extended encoding. This is done for both SAT solvers and for each grid size from 4 up to and including 7. For each puzzle generated, the time it took to do so is recorded. Finally, the minimum and maximum times, mean and standard deviation can be calculated for each combination of SAT solver and grid size to compare the times and draw conclusions from the measurements. This is script is repeated with an encoding where the clues are encoded directly.

4.1.2 Results

The results of the experiments can be found in tables 4.1 and 4.2.

Encoding	Size	Solver	Mean	Standard Deviation	Minimum	Maximum
Minimal	4	Glucose	0.044093	0.013651	0.025915	0.080267
Minimal	4	Kissat	0.037857	0.011326	0.023430	0.071046
Extended	4	Glucose	0.032511	0.006023	0.025495	0.067849
Extended	4	Kissat	0.037757	0.010783	0.024371	0.066477
Minimal	5	Glucose	0.207465	0.029993	0.143469	0.277037
Minimal	5	Kissat	0.199486	0.023828	0.145447	0.267250
Extended	5	Glucose	0.174917	0.021184	0.133067	0.234295
Extended	5	Kissat	0.187751	0.022598	0.151924	0.241726
Minimal	6	Glucose	2.754614	0.240025	2.479414	4.724840
Minimal	6	Kissat	2.763092	0.163215	2.318060	3.183430
Extended	6	Glucose	2.444797	0.142189	2.073514	2.841196
Extended	6	Kissat	2.739109	0.167014	2.407224	3.314149
Minimal	7	Glucose	52.451557	2.641039	46.252656	59.012146
Minimal	7	Kissat	57.787661	2.908416	50.681629	64.425849
Extended	7	Glucose	48.558272	2.513411	42.755939	55.184946
Extended	7	Kissat	57.526666	2.673567	51.498779	64.626179

Table 4.1: Results of experiments with different encodings, sizes and SAT solvers with the consequences of the clues encoded

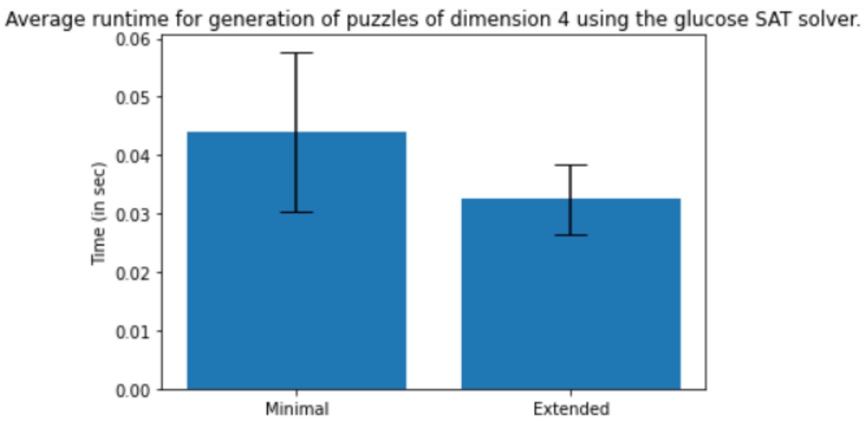
Encoding	Size	Solver	Mean	Standard Deviation	Minimum	Maximum
Minimal	4	Glucose	0.041908	0.012275	0.029860	0.066387
Minimal	4	Kissat	0.033121	0.007499	0.026219	0.064325
Extended	4	Glucose	0.033220	0.006481	0.029999	0.065829
Extended	4	Kissat	0.037152	0.009260	0.027156	0.058151
Minimal	5	Glucose	0.211810	0.026553	0.161693	0.272114
Minimal	5	Kissat	0.199394	0.018347	0.166830	0.255759
Extended	5	Glucose	0.189218	0.018899	0.161481	0.240587
Extended	5	Kissat	0.192154	0.020072	0.156994	0.241327
Minimal	6	Glucose	2.484660	0.134795	2.260973	2.897709
Minimal	6	Kissat	2.575514	0.130454	2.265213	2.840498
Extended	6	Glucose	2.378198	0.110093	2.126742	2.673342
Extended	6	Kissat	2.580993	0.130648	2.221307	2.983635
Minimal	7	Glucose	47.377423	2.077851	42.394269	53.133937
Minimal	7	Kissat	49.298266	2.407419	43.989253	56.028692
Extended	7	Glucose	43.873165	1.954557	40.401035	49.714977
Extended	7	Kissat	49.777189	2.054759	44.128048	54.358896

Table 4.2: Results of experiments with different encodings, sizes and SAT solvers with the clues encoded directly

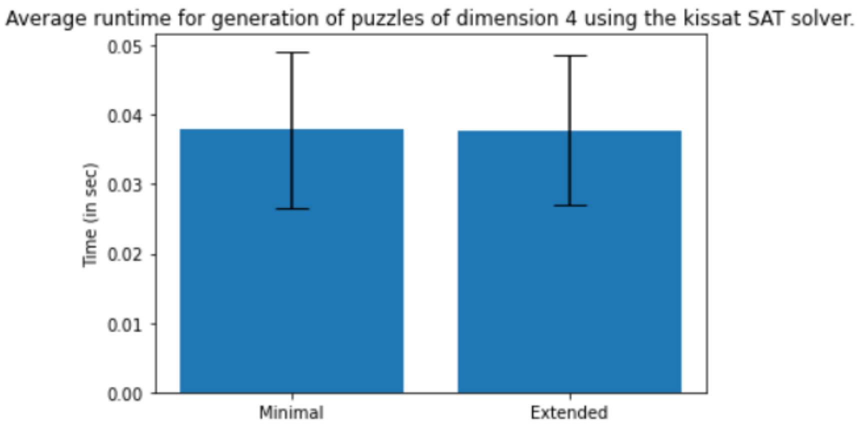
Minimal and Extended Encoding

To get more insight in the differences between using the minimum and the extended encoding in combination with the same SAT solver for the different sizes of puzzles, the mean and standard deviation for the results in table 4.1 are plotted in bar plots in figure 4.1.

As shown in both table 4.1 and figure 4.1, the extended encoding does have a (limited) positive impact on the mean time required to generate the puzzles. The difference is most noticeable for the puzzles generated with the Glucose SAT solver, while the time for the puzzles generated with the Kissat SAT solver remains almost the same. A possible explanation for this could be that the optimizations applied by the Glucose SAT solver benefit more from the extra clauses than those of the Kissat SAT solver.

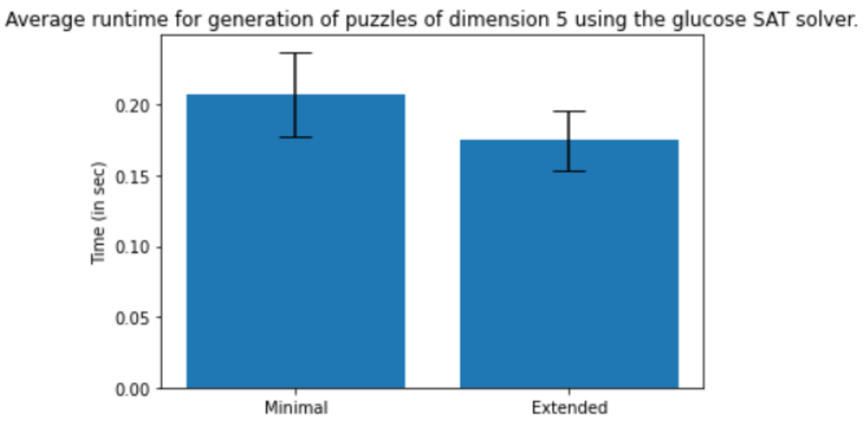


(a)

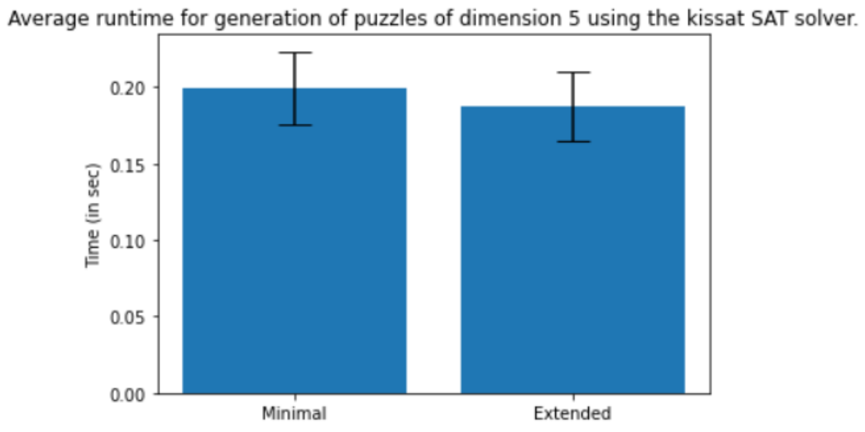


(b)

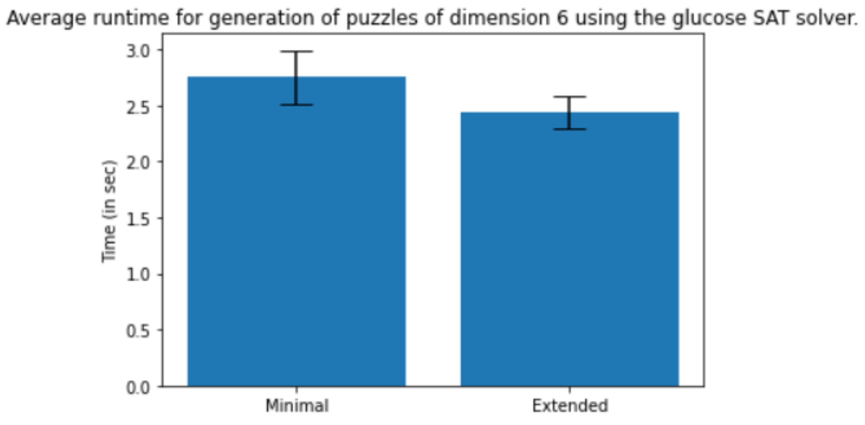
Figure 4.1: The mean and standard deviation of the generation time of puzzles for the minimal and extended encoding plotted against each other in a bar plot



(c)

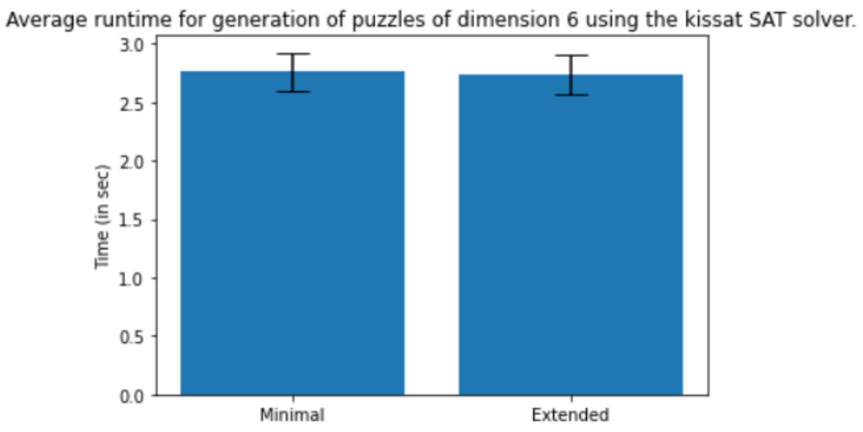


(d)

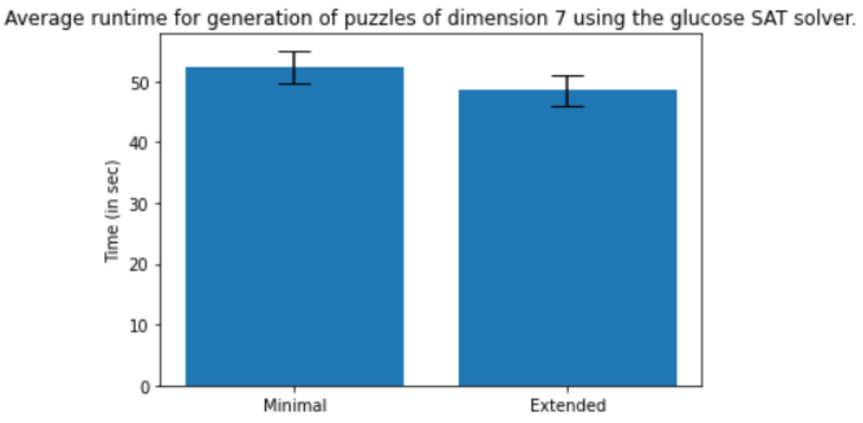


(e)

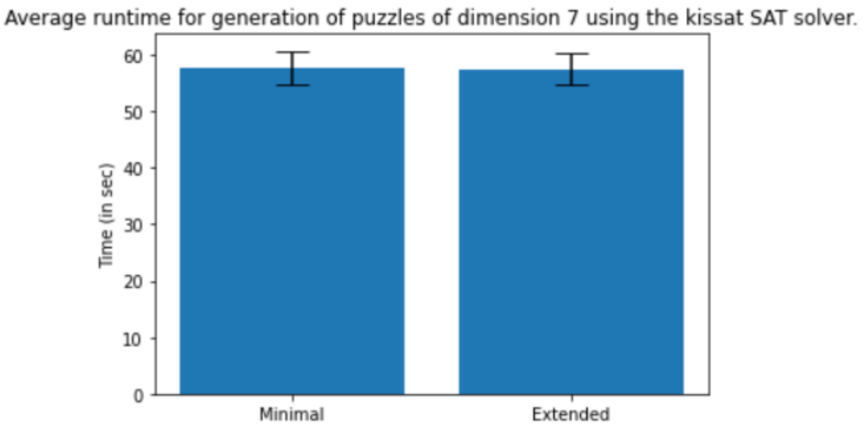
Figure 4.1: The mean and standard deviation of the generation time of puzzles for the minimal and extended encoding plotted against each other in a bar plot (cont.)



(f)



(g)



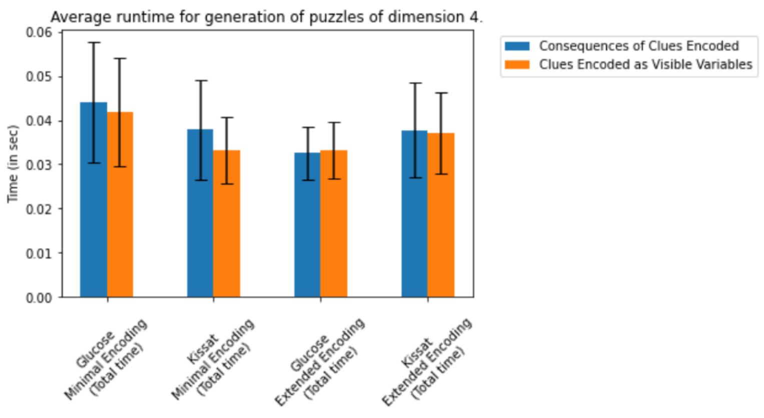
(h)

Figure 4.1: The mean and standard deviation of the generation time of puzzles for the minimal and extended encoding plotted against each other in a bar plot (cont.)

Different Encodings for Clues

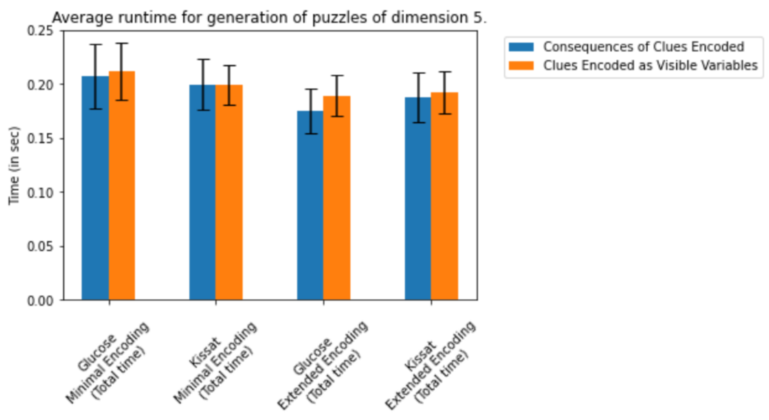
To get more insight in the differences in performance for the two SAT solvers when using each of the two encodings for clues, the mean and standard deviation of the generation times that can be found in tables 4.1 and 4.2 are plotted in bar plots in figure 4.2.

As shown in tables 4.1 and 4.2 and figure 4.2, the differences in generation time between the two ways of encoding the clues are negligible for puzzles of sizes 4, 5 and 6. For puzzles of size 7 directly encoding the clues as `visible` variables rather than encoding the consequences of the clues seems to be more efficient. Directly encoding the clues as `visible` variables adds more clauses to the CNF encoding, since for clues of 1 and n the `visible` variable is not needed when encoding the consequences of the clues. Similarly to the extended encoding for the Latin Square, the larger number of clauses could be a potential explanation for the decrease in generation times when encoding the clues directly as `visible` variables as this could allow the SAT solver to apply more optimizations. In addition, it could be that it takes less time to compute the clauses that are needed for the encoding where clues are directly encoded as `visible` variables as opposed to the encoding where the consequences of the clues are encoded.

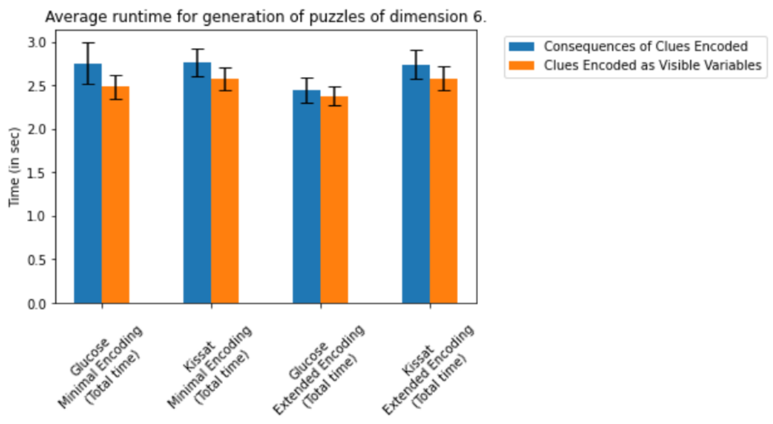


(a)

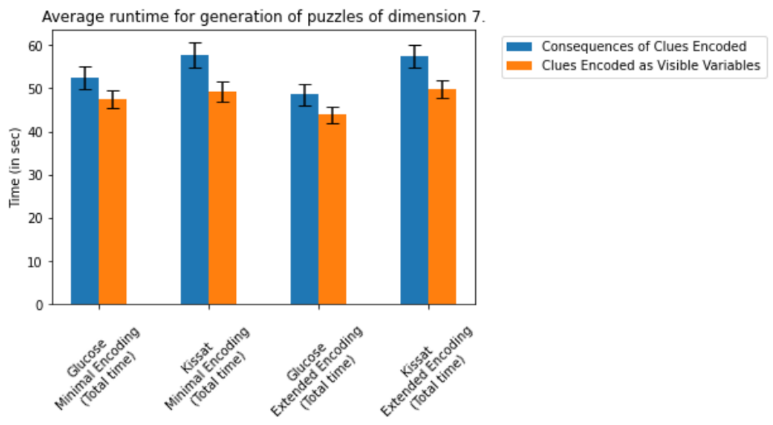
Figure 4.2: The mean and standard deviation of the generation time of puzzles for the different ways of encoding the clues plotted against each other in a bar plot



(b)



(c)



(d)

Figure 4.2: The mean and standard deviation of the generation time of puzzles for the different ways of encoding the clues plotted against each other in a bar plot (cont.)

Glucose SAT Solver and Kissat SAT Solver

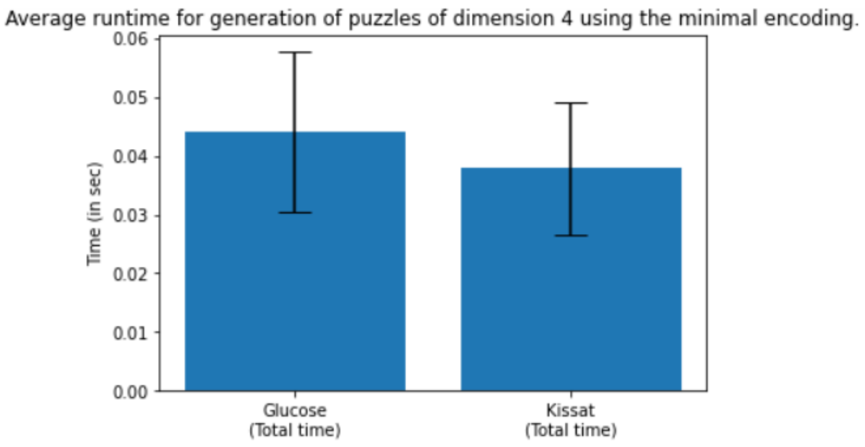
To get more insight in the differences between using the Glucose SAT solver and the Kissat SAT solver to generate the puzzles of different sizes, the mean and standard deviation of the generation times are plotted in bar plots in figure 4.3.

As shown in both table 4.1 and figure 4.3, in most cases the puzzles are generated slightly faster using the Glucose SAT solver than using the Kissat SAT solver. This could be caused by the Kissat SAT solver working less efficiently than the Glucose SAT solver for this particular problem. However, as visible in figure 4.4, in the puzzles generated with the Kissat solver more numbers are fixed. With the knowledge of how the puzzles are generated (section 3.4), it can be inferred that there is thus the need to make adjustments to the encoding more often and consequently to run the SAT solver more often. Since this all adds to the time to generate the puzzle, it cannot be said with certainty which SAT solver works better for this problem.

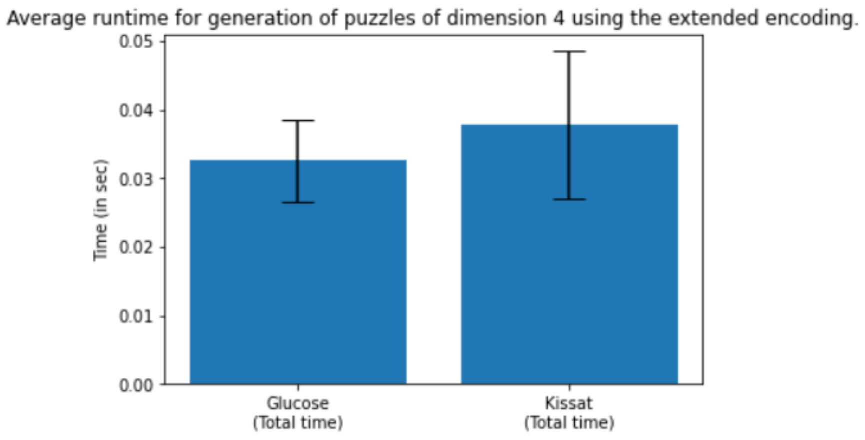
Looking at the puzzles[1] that are generated with both SAT solvers, it can be noted that a larger number of unique puzzles are generated using the Kissat SAT solver than using the Glucose SAT solver. Even when the puzzles are unique, the solutions generated with the Glucose SAT solver tend to overlap partially and the differences for the remainder of the solution are largely limited to shifting numbers by one or two cells. In contrast, in solutions generated with the Kissat SAT solver, there is virtually no overlap and the numbers are much more shuffled. A possible explanation for this could be that the randomness applied by the Glucose SAT solver is less than the randomness applied by the Kissat SAT solver.

It is to be expected that for similar solutions the amount of numbers to be fixed will be similar, while for completely different solutions the amount of numbers to be fixed is more likely to vary more. This could be a possible explanation for the fact that for some of the puzzles generated with the Kissat SAT solver, in comparison to puzzles generated with the Glucose SAT solver, a larger amount of numbers are fixed.

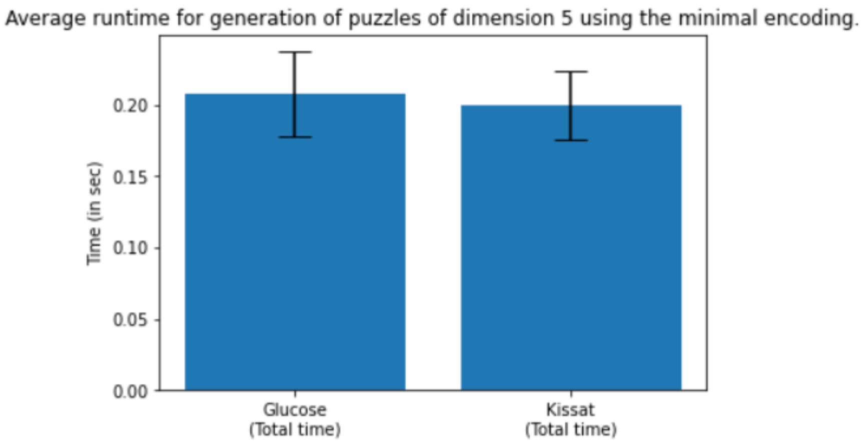
[1]These can be found in the folder "puzzles" at `https://gitlab.science.ru.nl/lkolijn/Skyscrapers`.



(a)

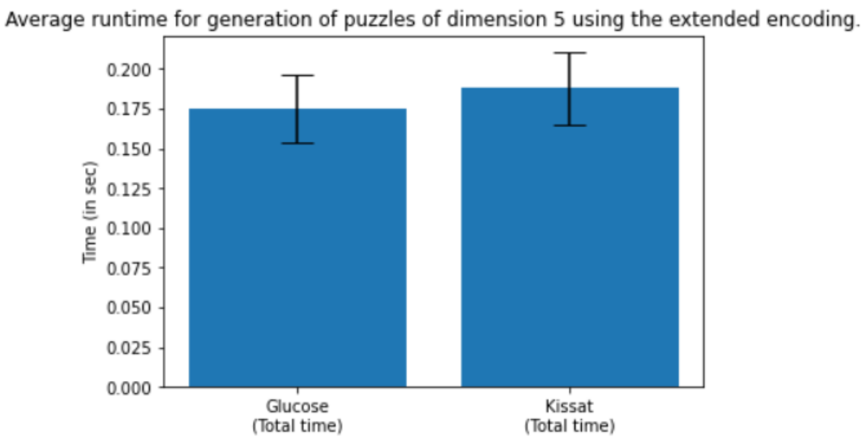


(b)

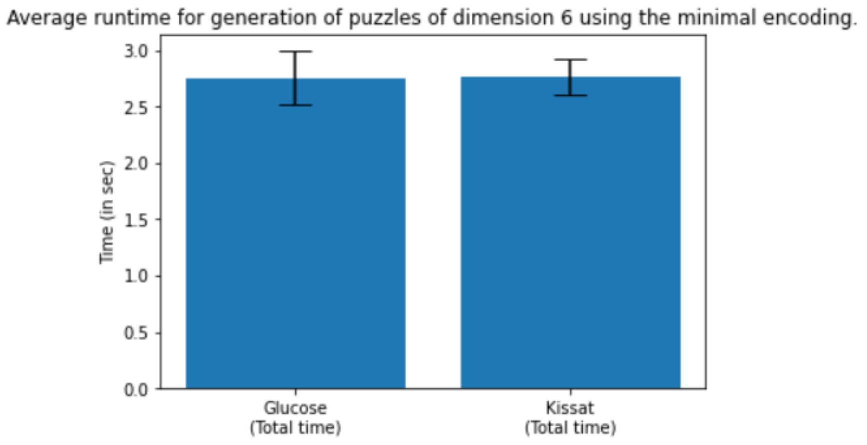


(c)

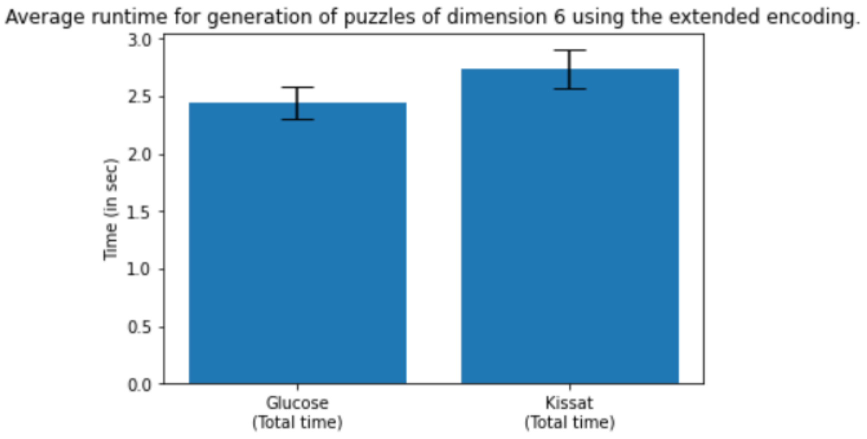
Figure 4.3: The mean and standard deviation of the generation time of puzzles for the Glucose and Kissat SAT solver plotted against each other in a bar plot



(d)

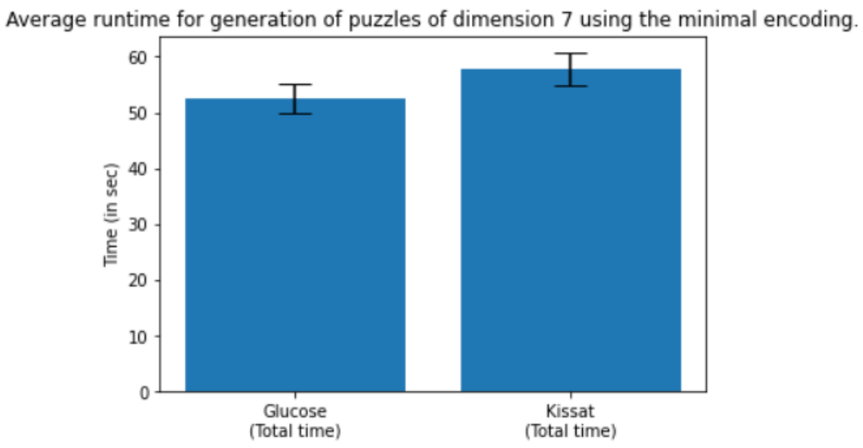


(e)

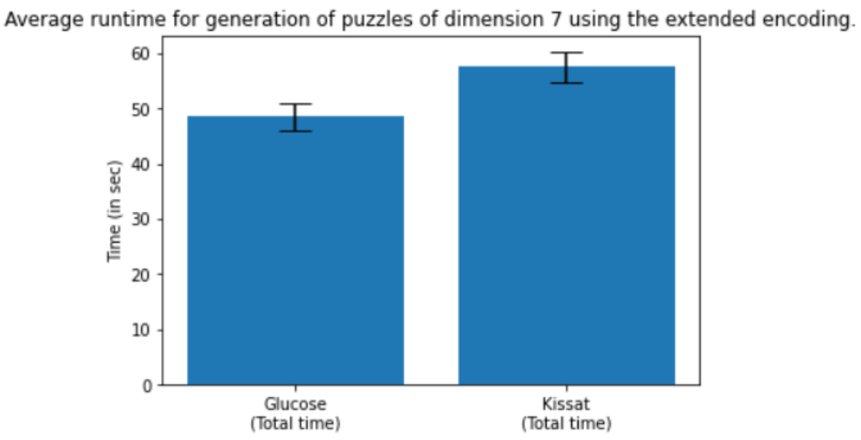


(f)

Figure 4.3: The mean and standard deviation of the generation time of puzzles for the Glucose and Kissat SAT solver plotted against each other in a bar plot (cont.)



(g)



(h)

Figure 4.3: The mean and standard deviation of the generation time of puzzles for the Glucose and Kissat SAT solver plotted against each other in a bar plot (cont.)

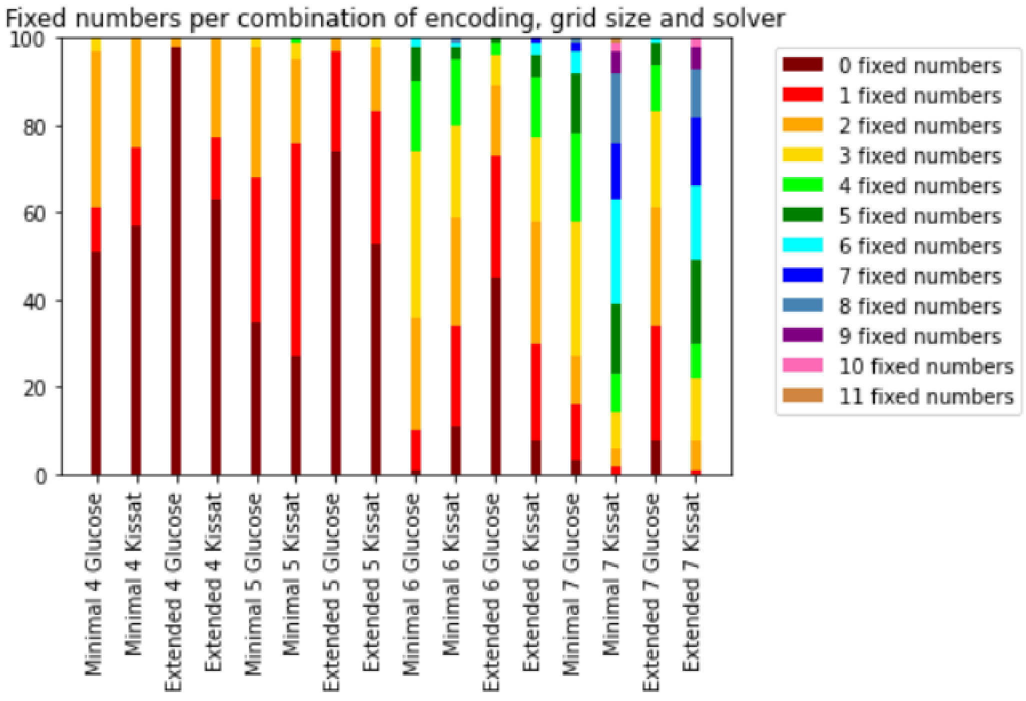


Figure 4.4: Distribution of the number of fixed numbers per combination of puzzle size, encoding and SAT solver

Chapter 5

Related Work

In 1971, SAT was the first problem that was classified as NP-complete by Stephen A. Cook [6]. Since then, a lot of research has been conducted on how to solve SAT problems, the applications of SAT, encoding other NP-complete problems as a SAT problem etc. A lot of the work that has been done, can be found in the Journal on Satisfiability, Boolean Modeling and Computation (JSAT) [1].

Every year there is a competition for SAT solvers, the purpose of which is “to identify new challenging benchmarks and to promote new solvers (. . .) as well as compare them with state-of-the-art solvers” [2]. Every year there are new and better performing SAT solvers, which also means that these solvers have more applications. An example of such a new application is the usage of SAT solvers in cryptanalysis [15].

A lot of research on the subject of logic puzzles has been performed on Sudoku puzzles, since that is arguably the most well-known logic puzzle. There are multiple papers on the topic of solving Sudoku puzzles as a SAT problem [11] [14] [22] and optimizing the SAT encoding [10].

Next to the Sudoku puzzle, there are several other puzzles for which there are papers on how to solve these puzzles as a SAT problem. Among these are the Fill-a-Pix puzzle [12], the Binary puzzle [20] and Flood-It [21]. The only research that was found on the Skyscrapers puzzle was research concerning the computational complexity of the puzzle [8] [9].

The Skyscraper puzzle, however, is different from the other puzzles that were mentioned before. For the Skyscraper puzzle there are not only the values in the grid that have to be taken into account, but also the clues that are located around the grid.

In addition to Boolean satisfiability problems, there is also a similar problem, the satisfiability modulo theory (SMT) problem. SMT problems can generalize SAT problems to more complex formulas in which all kinds of data structures can be used, such as integers, lists, etc. For the binary puzzle, this technique has already been used to solve the puzzle [19].

Chapter 6

Conclusion and Future Work

This thesis described how a Skyscrapers puzzle can be encoded as an SAT problem. This SAT encoding was then used to both solve Skyscrapers puzzles and generate new puzzles. From the experiments, it became clear that the extended encoding provides a small advantage in time over the minimal encoding for the Glucose SAT solver. For the Kissat SAT solver, this effect was not evident. In addition, regardless of the type of SAT solver and the type of encoding for the Latin Square, puzzle generation appears to be more efficient when clues are directly encoded as `visible` variables rather than encoding the consequences of the clues. Furthermore, it seemed that the Glucose SAT solver works a bit more efficient for this particular problem. However, full clarity could not be given on this because more numbers were fixed in the puzzles generated by the Kissat SAT solver, making it difficult to draw conclusions from the measurements as to a difference in the efficiency of the two SAT solvers.

As could be seen, the generator is not efficient when it comes to generating puzzles with $n \geq 7$. Future research could be performed to optimize the CNF encoding for the Skyscrapers puzzle such that generating bigger puzzles becomes more feasible. One possibility for might be to use other data structures to store the encoding or possibly use other variables.

In addition, all puzzles that are generated, have a full set of clues around the board. Further research could be done to find out how many clues are needed on average for a field of a certain size. If less than the full set of clues is placed around the board, this may also work in favor of generating larger puzzles. Fewer clues would mean that fewer variables need to be included in the encoding and therefore less time would likely need to be spent converting the puzzle to an encoding.

Another matter that can be addressed in further research is the fact that for the experiments the clues were computed for a newly generated solution every time. Therefore, the results of both SAT solver cannot be compared one to one. To be able to make a fair comparison, it might be an option to generate one hundred different sets of fields with the clues associated with them and use these sets for every combination of encoding and SAT solver. By doing so, a possible difference in the amount of randomness applied by both SAT solvers does not affect the initial configurations for the puzzles and a fairer comparison between the performance of the different SAT solvers can be made.

Finally, a starting point for more research can also be to investigate whether and how the skyscrapers puzzle could be rewritten to a satisfiability modulo theory (SMT) problem and whether an SMT solver can solve this problem more efficiently than an SAT solver.

Bibliography

[1] Journal on Satisfiability, Boolean Modeling and Computation. https://content.iospress.com/journals/journal-on-satisfiability-boolean-modeling-and-computation/12/1.

[2] The international Sat Competition web page. http://www.satcompetition.org/.

[3] Armin Biere et al. The Glucose SAT Solver. http://fmv.jku.at/kissat/.

[4] Alex Bellos. Can you solve it? Rise to the Skyscrapers challenge. https://www.theguardian.com/science/2018/jul/30/can-you-solve-it-rise-to-the-skyscrapers-challenge, July 2018.

[5] Curtis Bright, Jürgen Gerhard, Ilias Kotsireas, and Vijay Ganesh. Effective Problem Solving Using SAT Solvers. In *Communications in Computer and Information Science*. Springer International Publishing, 2020.

[6] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. ACM Press, 1971.

[7] Carla P. Gomes and David Shmoys. Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In *Proceedings of Computational Symposium on Graph Coloring and Generalizations*, 2002.

[8] Kazuya Haraguchi and Ryoya Tanaka. The Building Puzzle Is Still Hard Even in the Single Lined Version. *Journal of Information Processing*, 25(0), 2017.

[9] Chuzo IWAMOTO and Yuta MATSUI. Computational Complexity of Building Puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E99.A(6), 2016.

[10] Gihwon Kwon and Himanshu Jain. Optimized CNF Encoding for Sudoku Puzzles, 2006.

[11] Inês Lynce and Joël Ouaknine. Sudoku as a SAT Problem. In *Proceedings of International Symposium on Artificial Intelligence and Mathematics*, 2006.

[12] Aye Myint Myat, Khine Khine Htwe, and Nobuo Funabiki. Fill-a-Pix Puzzle as a SAT Problem. In *2019 International Conference on Advanced Information Technologies (ICAIT)*. IEEE, November 2019.

[13] Eén Niklas and Sörensson Niklas. The MiniSat Page. `http://minisat.se/`.

[14] Uwe Pfeiffer, Tomas Karnagel, and Guido Scheffler. A Sudoku-Solver for Large Puzzles using SAT. In *LPAR-17-short. short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning.*, volume 13 of *EPiC Series in Computing*. EasyChair, 2013.

[15] A. Ramamoorthy and P. Jayagowri. The state-of-the-art Boolean Satisfiability based cryptanalysis. *Materials Today: Proceedings*, July 2021.

[16] K.H. Rosen and K. Krithivasan. *Discrete Mathematics and Its Applications*. McGraw-Hill, seventh edition, 2013.

[17] Laurent Simon and Gilles Audemard. The Glucose SAT Solver. `https://www.labri.fr/perso/lsimon/glucose/`.

[18] Simon Tatham. Towers. `https://www.chiark.greenend.org.uk/~sgtatham/puzzles/`.

[19] Putranto Utomo. Satisfiability modulo theory and binary puzzle. *Journal of Physics: Conference Series*, 855, 06 2017.

[20] Putranto Utomo and Ruud Pellikaan. Binary puzzle as a SAT problem. In *Proceedings of the 2017 Symposium on Information Theory and Signal Processing in the Benelux*, May 2017.

[21] Milan van Stiphout. Flood-It as a SAT problem. Bachelor's thesis, Radboud University, 2020.

[22] Tjark Weber. A SAT-based Sudoku Solver. In *Proceedings of The 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2005.