Opleiding Informatica

Efficiency of two SAT encodings
of Yajilin puzzles

Bryan Kaak

Supervisors:
Hendrik Jan Hoogeboom & Jeannette de Graaf

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

26/07/2025

Abstract

Yajilin is a logic puzzle played on a two-dimensional grid, where black cells have to be placed on the board in order to create a single cycle that passes only through white cells. In this thesis, we apply two methods to solve these puzzles. Both methods use propositional logic, and we solve the puzzle using SAT solvers. The first method uses a counting method to construct the cycle. The second method uses two different trees to construct the cycle. In the end, we compare the two methods and see which one is more efficient when solving the puzzles.

Contents

1 Introduction

Logic puzzles are a great domain to apply the power of SAT solvers. In this case, we will use the puzzle Yajilin. This puzzle, created by the company Nikoli [Nik], consists in the beginning of an $n \times m$ grid with cells containing either a number with a corresponding direction (an indicative cell) or white cells. The number indicates how many black cells there are in the given direction. The first goal of the puzzle is to place the black cells based on the constraints provided by the indicative cells. However, there may exist black cells which are not indicated by an indicative cell. The second goal is to construct a single cycle that passes through all remaining white cells. This cycle connects the white cells only through horizontal or vertical connections. Figure 1 shows an example of the puzzle with a correct and an incorrect solution. The player must follow the following rules [JJb]:

- Every indicative cell with number k and direction d must have exactly k black cells in direction d.
- There may exist black cells that are not indicated by indicative cells.
- Indicative cells are never black.
- Black cells do not touch each other horizontally or vertically.
- Construct a cycle that passes all white cells.
- The cycle passes through each white cell exactly once: it does not cross or touch itself.
- The cycle passes horizontally or vertically through the centre of the white cells.

When solving these puzzles, you can find patterns to get to the solution. For example, next to every black cell, there must be a part of the cycle or an indicative cell. As far as we know, every puzzle of Yajilin has a unique solution. Furthermore, the Yajilin puzzle is proven NP-complete using a reduction from a modified version of the Hamilton cycle problem on planar undirected graphs [ISI12].

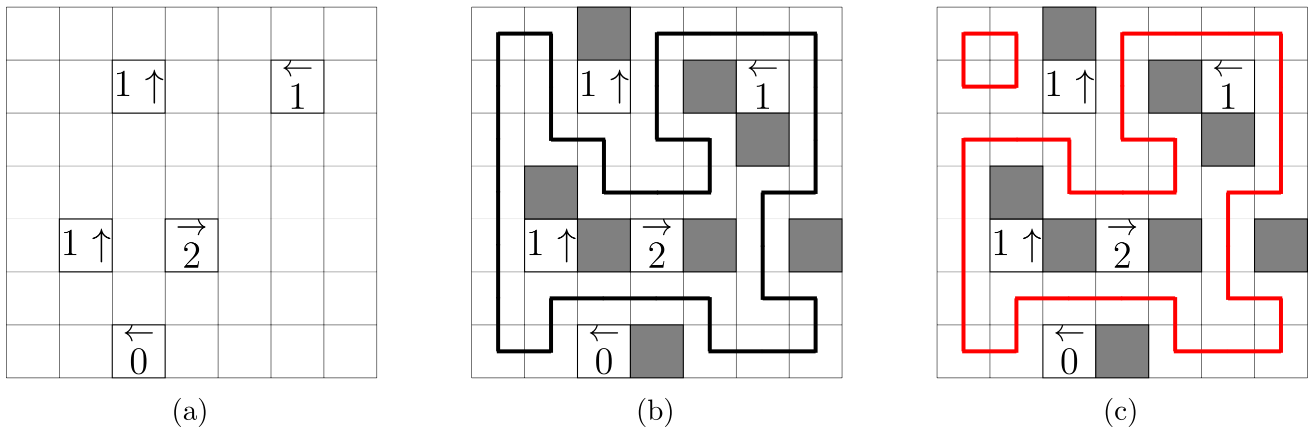

Figure 1: **(a)** Example puzzle of Yajilin. **(b)** The correct solution of the puzzle. **(c)** An incorrect solution of the puzzle. [Nik]

1.1 Thesis overview

The current chapter introduces the Yajilin puzzle and explains its rules. Chapter 2 discusses the background of translating the rules of a puzzle into propositional formulas. It also introduces the research question. The background is partially used in Chapter 3 to construct formulas for the Yajilin puzzle. In that chapter, we explain the various formulas and the program we made. Chapter 4 contains a description of the experiments that were performed and the results of comparing the two methods. Finally, Chapter 5 concludes the thesis and gives ideas for future work.

The supervisors of this thesis are Hendrik Jan Hoogeboom and Jeannette de Graaf, associated with the Leiden Institute of Advanced Computer Science (LIACS).

2 Related Work & Background

2.1 Propositional logic

To solve the Yajilin puzzle, we make extensive use of propositional logic. We construct propositional formulas using logical operators such as AND, OR, NOT and IMPLIES ($\wedge, \vee, \neg, \rightarrow$). These operators can be used to express a relationship between variables. For example, given variables x, y and z, we can construct different formulas. Formula 1 is a simple example. This formula evaluates to true when both x and y are true or z is true. Formula 2 is a second example. This formula sets z to true, as long as x and y are true.

$$(x \wedge y) \vee z \tag{1}$$

$$(x \wedge y) \rightarrow z \tag{2}$$

It is also possible to use the big AND or OR operators ($\bigwedge, \bigvee$). These operators indicate that a formula is repeated using conjunctions or disjunctions. This prevents writing out long and complex formulas. Additionally, these operators can also include boundaries. For example, $\bigwedge_{i=1}^{3} y_i \vee z_i$, denotes a conjunction of three formulas, namely $(y_1 \vee z_1) \wedge (y_2 \vee z_2) \wedge (y_3 \vee z_3)$.

Since we use SAT solvers to solve puzzles, we need to convert the rules into Conjunctive Normal Form (CNF). A formula is in CNF if it consists of several subformulas, which only use the OR operator. These subformulas are combined with the AND operator. We call each subformula a clause. Formula 3 is an example of a formula that is in CNF.

$$(x \vee y) \wedge (\neg x \vee y \vee z) \wedge z \tag{3}$$

2.2 Converting rules to formulas

We have to count the black cells to fulfill the first rule of the Yajilin puzzle, as some cells have numerical constraints. There must be k black cells in the region of the indicative cell. There are several methods to translate these requirements into propositional formulas.

One of the methods used to solve the numerical constraint is applied to the Juosan puzzles by Ammar et al. [AAW24]. In this puzzle, we have to count the number of horizontal and vertical stripes in a region. The number of these stripes must be equal to the number s, indicated by the region. Ammar et al. [AAW24] have translated this problem into propositional formulas. They check whether the number given by the constraint equals the number of horizontal or vertical stripes. They define two formulas to check this equality, one that checks that at least s cells have horizontal or vertical stripes and one that checks that at most s cells have horizontal or vertical stripes. By combining these formulas, they enforce that exactly s cells contain horizontal or vertical stripes. For the Yajilin puzzle, we will use a similar method to count the number of black cells in a certain direction.

2.3 Connectivity Constraint

The Yajilin puzzle is a logic puzzle with a connectivity constraint. The solution consists of a single cycle that contains all adjacent white cells. We consider two methods to translate the connectivity constraint into a propositional formula. The following two subsections discuss these two methods.

2.3.1 Classic Method

The first method is most commonly used for puzzles that include a connectivity constraint. Gerhard van der Knijff developed a general method to translate this constraint into propositional formulas [vdK21]. He uses the following theorem to achieve this:

Theorem 1 *Graph (V,E) is connected if and only if $f : V \to Int$ exists such that for all $i \in V \backslash \{1\}$ a node $j \in V$ exists such that $(i, j) \in E$ and $f(j) < f(i)$*

In our case, the graph represents (part of) the grid, where $V \subseteq \{1, \ldots, N\}$ corresponds to the cells of the board, where N is equal to the total number of cells ($N = n \times m$). This theorem states that a number can be assigned to each cell such that each cell has a neighbour with a lower number, except the starting cell, which contains the number 1 in this case. To ensure that the solution contains exactly one cycle, an additional constraint is required. Gerhard van der Knijff introduced this additional constraint by specifying that every cell must have exactly zero or two neighbours [vdK21]. This ensures that only cells with two neighbours are part of the cycle, preventing dead ends (which are cells with only one neighbour). This constraint, in combination with Theorem 1, guarantees that the solution contains exactly one cycle. However, this method has one drawback. Assigning numbers to the cells requires approximately $O(N^2)$ variables in total, where $N = n \times m$, the size of the board [Str25]. This method was also used by Niels Heslenfeld to solve the Fobidoshi puzzles [Hes24] and by Roos Wensveen to solve the Hitori puzzles [Wen24].

2.3.2 Novel Method

The second method was first used to demonstrate that picture-languages are recognisable [Rei98]. Klaus Reinhardt achieved this by constructing two trees. One tree covers the cells themselves, which also includes the solution. The second tree covers the corners of the cells. Klaus Reinhardt states that these two trees "completely intrude the spaces between the other tree and hereby avoid any cycle" [Rei98]. This method has rarely been applied to solve the connectivity constraint of logic puzzles. However, Hanna Straathof used this method for the Kuroshuto puzzles [Str25].

The idea behind this method is to assign a constant number of variables to each cell, resulting in a total of $O(N)$ variables [Hoo]. This is a significant improvement compared to the classic method, where $O(N^2)$ variables are needed. For this reason, it is worth investigating whether this novel method is also more efficient in solving such puzzles. This leads to the following research question:

How can different SAT encodings help in solving logic puzzles with connectivity constraints efficiently?

We would like to find the computational difference between the different methods. However, it is too complicated to test these encodings on a lot of different puzzles. Therefore, we test these encodings only on the Yajilin puzzle. This results in the following subquestion:

How can we solve the Yajlin puzzle using SAT encodings?

This subquestion helps us in finding the encodings needed to solve the Yajilin puzzle. After this, we can examine the different aspects of the encodings, such as the number of variables used and the runtime, to answer the main research question.

3 SAT Solver

We use SAT solvers to find solutions for Yajilin puzzles. These solvers work with SAT encodings, which consist of propositional formulas representing the rules and constraints of the puzzle. The SAT solver determines whether a given propositional formula is satisfiable or unsatisfiable. If it is satisfiable, the puzzle has a solution. This means that the SAT solver tries to find a way to assign Boolean values (true or false) to the variables such that all clauses are satisfied.

In this chapter, we will explain the different SAT encodings used for Yajilin puzzles. We begin by translating the general rules, such as finding the positions of the black cells. Next, we discuss the formulas used in the two methods for creating the solution's cycle. To ensure that the formulas are clear and explainable, not all of the formulas are presented in Conjunctive Normal Form (CNF). Details about how to convert formulas to this form can be found in Section 3.4. Section 3.5 specifies the initial indexes of each variable, and Section 3.6 describes the functionality of the program. Table 1 shows a description of all the variables used to solve the puzzle.

Variable	Description
Z_x	Cell x is black.
C_x	Cell x is an indicative cell.
$T_{v,t}$	Total number of black cells between boundary cell and cell v.
$E_{x,d}$	Outgoing edge from cell x to neighbour d in the classic method.
$Q_{x,k}$	Cell x has number k in the classic method.
$R_{x,d}$	Outgoing edge in cell tree from cell x to neighbour d in the novel method.
$H_{x,d}$	Outgoing edge in corner tree from cell x to neighbour d in the novel method.

Table 1: Description of each variable.

3.1 General Rules

The Yajilin puzzle is defined on a board of size $n \times m$. Every cell x on this board is assigned two variables, Z_x and C_x. The variable Z_x is true if cell x is black, otherwise it is false. The variable C_x indicates whether cell x is an indicative cell.

To ensure that the SAT solver correctly places the black cells, we surround the original board with boundary cells. Figure 2 illustrates this construction, where each red asterisk ($*$) represents a boundary cell. These boundary cells are most of the time omitted from figures, unless they are necessary for clarity.

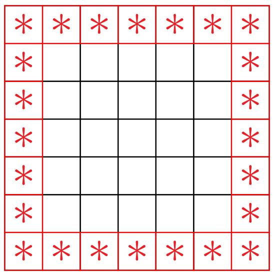
Figure 2: Placement of boundary cells around the original board.

Every indicative cell defines its own region, consisting of all cells between the indicative cell and the boundary cell in the direction specified by the indicative cell. To distinguish cells within these regions from cells of the entire board, we refer to a cell in a region as cell v. Although SAT solvers cannot count directly, they can still determine the number of black cells in a region. This is done by assigning $k + 1$ variables to each cell in the region, including boundary and indicative cells. Here, k denotes the number in the indicative cell, specifying the number of black cells in that region. We denote these variables as $T_{v,t}$. This count variable represents the total number of black cells (t) between the boundary cell and cell v, including cell v. We start counting at the boundary cell, as shown in Figure 3, and proceed towards the indicative cell. This boundary cell is necessary because it is a cell that is never black. This ensures that we start with $t = 0$. If we start at the first non-boundary cell, then it can be either white or black, making the value of t unknown ($t = 0$ or $t = 1$). If cell v is black, we immediately increment t. In the end, t must equal k at the indicative cell.

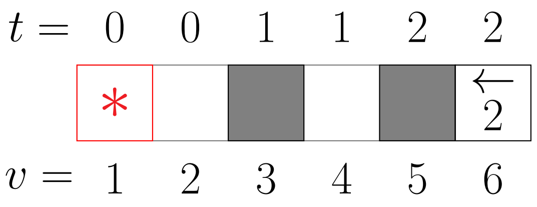

Figure 3: Method used for counting black cells.

Since we start counting at the boundary cell, all variables $T_{v,t}$ with $t \neq 0$ must be false, and the variable $T_{v,0}$ must be true. Formulas 4 and 5 enforce these constraints.

$$T_{v,0} \quad \text{if } v \text{ is a boundary cell} \tag{4}$$

$$\bigwedge_{t=1}^{k} \neg T_{v,t} \quad \text{if } v \text{ is a boundary cell} \tag{5}$$

Each cell between the indicative cell and the boundary cell is also assigned $k + 1$ count variables. For every white cell, exactly one of these count variables should be true. However, it is sufficient to enforce that at least one is true. The reason for this will be explained later in this section. Formula 6 is used to declare the count variables for white cells in each region.

$$\bigvee_{t=0}^{k} T_{v,t} \quad \text{if } v \text{ is a white cell} \tag{6}$$

The indicative cell also has special constraints. For an indicative cell v with value k, the count variable $T_{v,k}$ must be true. This k represents the number of black cells in the region of the indicative cell. Formula 7 declares which variable must be true, while formula 8 declares the other variables, where $t \neq k$, as false.

$$T_{v,k} \quad \text{if } v \text{ is an indicative cell} \tag{7}$$

$$\bigwedge_{t=0}^{k-1} \neg T_{v,t} \quad \text{if } v \text{ is an indicative cell} \tag{8}$$

Now that all count variables are declared, we can determine when a specific count variable should be set to true. There are two cases to consider, depending on whether the current cell v is white ($\neg Z_v$) or black (Z_v). First, if the current cell v is white, then formula 9 applies. In this case, the count variable is copied from the previous cell $v-1$ because no new black cell has been encountered. Second, formula 10 describes the case in which cell v is black. In this case, the count variable must be incremented, as a new black cell has been encountered. This is done by increasing the value of t from the previous cell $v-1$ by one. These two formulas are applied for each value t to determine which count variable must be set to true. Together with the constraint that the count variable $T_{v,k}$ must be true at the indicative cell, these formulas ensure that exactly one count variable is true in each cell. Because the cell v before the indicative cell must have either the count variable $T_{v,k-1}$ or $T_{v,k}$ set to true, depending on whether the current cell is black. This derivation is passed to the other cells in the region, ultimately ensuring that exactly one count variable is true in each cell.

$$\bigwedge_{t=0}^{k} [(\neg Z_v \wedge T_{v-1,t}) \rightarrow T_{v,t}] \quad \text{for } v \neq \text{boundary} \tag{9}$$

$$\bigwedge_{t=0}^{k} [(Z_v \wedge T_{v-1,t}) \rightarrow T_{v,t+1}] \quad \text{for } v \neq \text{boundary} \tag{10}$$

The formulas used so far do not strictly enforce the required number of black cells. It is still possible to indirectly count beyond the value k. If we are currently at cell v, and the count variable $T_{v-1,k}$ was true in the previous cell, then the SAT solver might still make the current cell black, incrementing t to $k+1$. However, the variable $T_{v,k+1}$ does not exist. The variable $T_{v,k+1}$ would correspond to the first count variable of the next cell, which is not the intended behaviour. This behaviour occurs due to the technical constraints of SAT solvers. Internally, SAT solvers place the variables sequentially and use their indices to reference them. As a result, incrementing the index of a variable by one results in referring to the wrong variable. To avoid this problem, we make use of formula 11. This formula ensures that if the count variable with $t = k$ is already true in the previous cell, then the current cell cannot be black. So, once the required number of black cells has been reached, no additional black cells are allowed in that region.

$$T_{v-1,k} \rightarrow \neg Z_v \tag{11}$$

At this point, exactly k black cells exist in the region of each indicative cell. However, it is still possible that two black cells are adjacent to each other, which is not allowed by the rules of the puzzle. To avoid this, we use formula 12. Since black cells can appear both within the regions and outside the regions (i.e. black cells not indicated by indicative cells, whose positions also depend on the cycle), we use another variable to represent the cells of the board. In this formula, x denotes a cell on the board instead of a cell in a region. Variable d indicates a neighbour of cell x. The variable d can take four values, 0, 1, 2, and 3. These values represent the neighbouring cell in the directions north, east, south and west, respectively. The formula ensures that if cell x is black, none of its neighbours are also black.

$$Z_x \rightarrow \bigwedge_{d=0}^{3} \neg Z_d \tag{12}$$

3.2 Classic Method

Now that the formulas satisfying the general rules have been constructed, we have to make formulas to construct a valid cycle. Figure 4 shows the approach used to construct the loop for the classic method. Although the cycle is undirected in the puzzle, we represent it as a directed cycle since it is easier to create formulas for a directed graph.

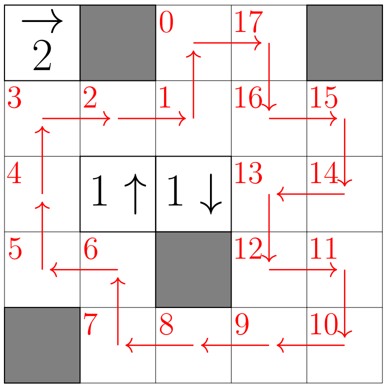

Figure 4: Example of the approach used to construct the cycle using the classic method.

First, we ensure that every white cell has exactly one outgoing and one incoming edge. This guarantees that every white cell has exactly two white neighbours, as required by Gerhard van der Knijff [vdK21]. These constraints can construct multiple cycles. To create a single cycle, we have to count in the same way as we did with the black cells. We start counting at the white starting cell x_0. This cell is chosen manually and will get the number 0. Every other white cell with two white neighbours must also get a number. It must hold that every cell, except the starting cell, has an outgoing edge to a cell with a lower number and that the cycle ends in the starting cell with number 0. This ensures that there are no multiple cycles possible, since any cycle not containing the starting cell cannot end at number 0, so making multiple cycles is impossible.

We will first discuss the outgoing edges and later the incoming edges of the cells. Since the cycle passes only through white cells, it is necessary to distinguish between white and black cells when constructing the cycle. Since we do not know which cells are black or white in advance, every white cell x is assigned four variables corresponding to its outgoing edges, one for each direction, denoted as $E_{x,d}$. This variable indicates whether there is an edge from cell x to its neighbour in direction d. Again, for simplicity, we use d to represent the neighbour of cell x in direction north, east, south or west (denoted by $0, 1, 2$ and 3). No edge variables are assigned to indicative cells, since we know beforehand that these cells are not part of the cycle. Furthermore, we skip the indicative cells in each formula of the classic method, since these cells are not important when constructing the cycle with the classic method. This way, we can use $\neg Z_x$ to represent a white cell (instead of using $\neg Z_x \wedge \neg C_x$). Additionally, no edge variables are declared for edges from a cell x towards a boundary cell, since edges cannot go to the border of the board. If cell x is black, it is not part of the cycle. Therefore, all the outgoing edges from this cell must be set to false. Formula 13 enforces this constraint.

$$Z_x \rightarrow \bigwedge_{d=0}^{3} \neg E_{x,d} \tag{13}$$

Determining the outgoing edges of a white cell is more complex. Each white cell must have exactly one outgoing edge to a neighbour. This neighbour must also be a white cell, as black and indicative cells are excluded from the cycle. To ensure that exactly one outgoing edge is true, we have to set at least one and at most one outgoing edge to true. Formula 14 ensures that at least one outgoing edge is true. Formula 15 ensures that at most one edge is set to true. This is done by examining each pair of edges and forbidding both from being true at the same time. Formula 14 and 15 together ensure exactly one outgoing edge. However, this does not guarantee that the outgoing edge is going to a white cell. For this, we use formula 16. This formula states that if the neighbour in direction d is a black or an indicative cell, the corresponding edge must be false. These three formulas together ensure that each white cell has exactly one outgoing edge to another white cell.

$$\neg Z_x \rightarrow \bigvee_{d=0}^{3} E_{x,d} \tag{14}$$

$$\begin{aligned} \neg Z_x \rightarrow [&\neg(E_{x,0} \wedge E_{x,1}) \wedge \neg(E_{x,0} \wedge E_{x,2}) \wedge \neg(E_{x,0} \wedge E_{x,3}) \wedge \\ &\neg(E_{x,1} \wedge E_{x,2}) \wedge \neg(E_{x,1} \wedge E_{x,3}) \wedge \neg(E_{x,2} \wedge E_{x,3})] \end{aligned} \tag{15}$$

$$(Z_d \vee C_d) \rightarrow \neg E_{x,d} \tag{16}$$

We must also ensure that every white cell has exactly one incoming edge. It is not necessary to check whether black or indicative cells receive incoming edges, since we have already forbidden outgoing edges from black cells and disallowed edges from white cells to black or indicative cells. So, it is impossible to have an incoming edge at black and indicative cells.

To ensure that each white cell has exactly one incoming edge, we use the same approach as we did for the construction of the outgoing edges. We require at least one and at most one incoming edge. We use a slightly different way of denoting the variables representing the incoming edges. The variable $E_{d,x}$ represents an edge directed from the neighbour of the current cell x in the direction d towards the current cell x. For each white cell, formulas 17 and 18 must hold. These formulas are the same as for outgoing edges, but change the point of view to the target cell. These two formulas, combined with the previous four formulas, ensure that cycles are created through white cells.

$$\neg Z_x \rightarrow \bigvee_{d=0}^{3} E_{d,x} \tag{17}$$

$$\begin{aligned}\neg Z_x \rightarrow [&\neg(E_{n,x} \wedge E_{e,x}) \wedge \neg(E_{n,x} \wedge E_{s,x}) \wedge \neg(E_{n,x} \wedge E_{w,x}) \wedge \\ &\neg(E_{e,x} \wedge E_{s,x}) \wedge \neg(E_{e,x} \wedge E_{w,x}) \wedge \neg(E_{s,x} \wedge E_{w,x})]\end{aligned} \tag{18}$$

To ensure that the solution forms exactly one cycle, we use a counting method. Each white cell x is assigned a set of count variables $Q_{x,k}$, where k ranges from 0 to $(n \times m) - 1$. So, each white cell has $n \times m$ variables in total. We also determine a white starting cell x_0, for which the count variable $Q_{x_0,0}$ is set to true, while all other count variables for this cell must be set to false. This is ensured by formulas 19 and 20. For all other white cells, the variable $Q_{x,0}$ is set to false and at least one of the other count variables is set to true. This is done by formulas 21 and 22.

$$Q_{x_0,0} \tag{19}$$

$$\bigwedge_{k=1}^{(n\times m)-1} \neg Q_{x_0,k} \tag{20}$$

$$\neg Q_{x,0} \quad \text{if } x \neq x_0 \tag{21}$$

$$\bigvee_{k=1}^{(n\times m)-1} Q_{x,k} \quad \text{if } x \neq x_0 \tag{22}$$

With the formulas above, we can assign one or more numbers to each cell. With this method, we are also assigning numbers to black cells. However, these numbers are disregarded when we evaluate whether the neighbour of a white cell has a lower number than the white cell itself. This does not cause any issues, since black cells are not part of the cycle and have no incoming edges.

For every white cell x, which is not x_0, we have to ensure that if there is an edge from cell x to its neighbour in direction d, then the count variable that is true for the neighbour must have a lower number than the one that is true for cell x. This is done by formula 23, which guarantees that exactly one count variable is set to true in each white cell. For example, if cell x has an edge to the starting cell x_0, then the only valid count variable for x is $Q_{x,1}$, since only $Q_{x_0,0}$ is true. As a result, all other count variables for cell x must be false. This derivation is passed through the cycle, ensuring that exactly one count variable is set to true for every white cell in the loop.

$$\bigwedge_{d=0}^{3} \bigwedge_{k=1}^{(n*m)-1} [E_{x,d} \to (Q_{x,k} \to Q_{d,k-1})] \quad \text{if } x \neq x_0 \tag{23}$$

3.3 Novel Method

As discussed before, the novel method constructs two trees. The first tree covers the cells of the puzzle and is referred to as the cell tree. The second tree covers the corners of the cells and will be called the corner tree. An example of how these trees are constructed in a Yajilin puzzle is shown in Figure 5. Here, the red tree represents the cell tree, and the blue tree represents the corner tree.

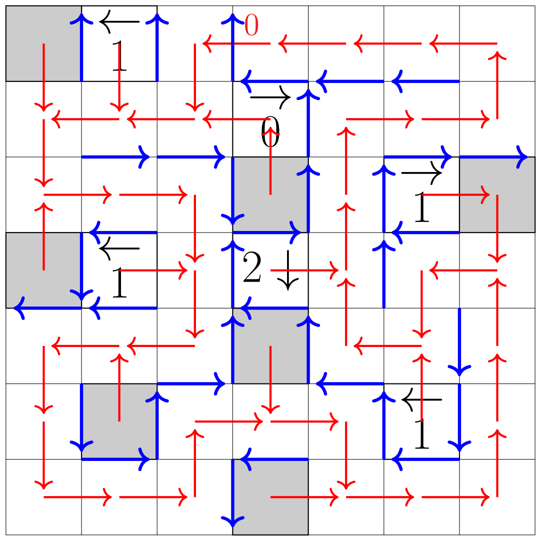

Figure 5: Example illustrating the construction of the cycle using the novel method.

In the original problem, discussed by Klaus Reinhardt, these two trees are constructed to ensure that no cycles are formed within both trees [Rei98]. However, the solutions of Yajilin puzzles must contain a cycle. We can solve this problem by disregarding one edge of the cell tree. We remove one edge from the cycle to transform it into a tree. For simplicity, and because the study focuses on the method of constructing the cycles, this edge is chosen manually beforehand by looking at the solution of the puzzle. We can look at the solution since we use puzzles from the Janko website [JJb]. With certain constraints in the cell tree, which we will discuss later, this omitted edge will be constructed but not checked when checking for crossing edges between the two trees. In practice, we disregard two edges, which we refer to as the starting edges. This is necessary because the cell tree is an undirected graph, and its direction is unknown beforehand. If we only define a single starting cell and disregard its outgoing edge, that edge might, for example, go south into the puzzle and not lie near the border of the board. This is not a valid starting edge. According to Hanna Straathof, cycles can be created around the starting cell when this starting cell does not lie near the border of the board [Str25]. That is why we choose the starting edges near the border of the board.

In this chapter, we will discuss the formulas describing the cell tree, the corner tree and the formulas involving both trees.

3.3.1 Cell tree

The construction of the cell tree is almost identical to the approach used to construct the cycle in the classic method. The cell tree is constructed as a cycle, but we will consider it as a tree later by disregarding the starting edges. The only big difference, compared to the classic method, is that black and indicative cells will also have an outgoing edge. This is necessary because forbidding outgoing edges from black or indicative cells can create cycles in the corner tree, as shown in Figure 6. Such cycles in the corner tree prevent the cell tree from being acyclic. The construction of these trees and how we can prevent acyclicity in both trees will be explained later.

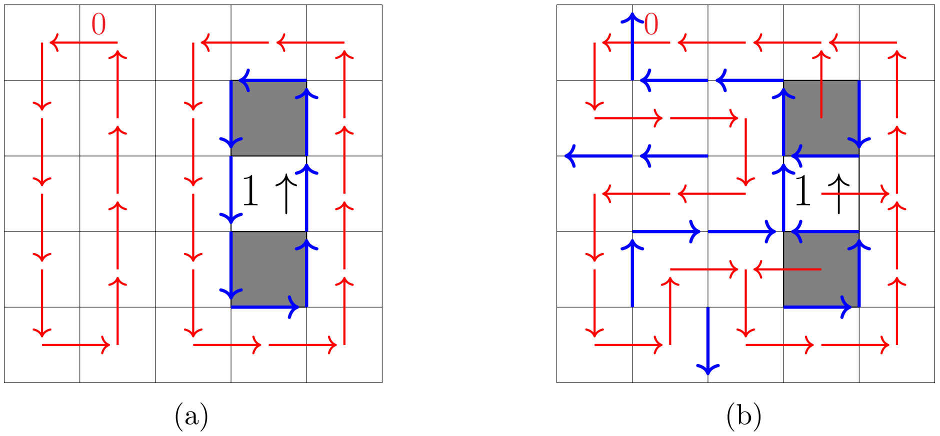

Figure 6: **(a)** An incorrect solution of the puzzle. **(b)** A correct solution of the puzzle.

For the cell tree, every cell on the board is assigned variables that represent its outgoing edges. As in the classic method, no variables are created for edges pointing to the boundary cells. These variables are denoted as $R_{x,d}$, they are true when there is an edge from the current cell x to its neighbour in direction d.

First, we require that every cell, including black and indicative cells, has exactly one outgoing edge. This requirement is easier to enforce than in the classic method, as we no longer need to verify whether a cell is white. To ensure exactly one outgoing edge per cell, we use the same approach as before. Formula 24 ensures that each cell has at least one outgoing edge, and formula 25 ensures that each cell has at most one outgoing edge. At the same time, this does not prevent edges from pointing towards black or indicative cells. This is a bit more complex to achieve, because unlike in the classic method, black and indicative cells are allowed to have an outgoing edge to another black or indicative cell. This is necessary because such cells might be surrounded by other black or indicative cells. If we prevent black and indicative cells from having edges towards other black and indicative cells, it is impossible to construct the cell tree. This is why we need to check if a cell is white in formula 26. In the classic method, this was not needed because indicative cells were not assigned edge variables. In the novel method, the indicative cells will get the edge variables assigned to them. A cell is considered white if it is neither black nor an indicative cell. If a cell x is white and its neighbour in direction d is a black or an indicative cell, then an edge in direction d is not allowed. This prevents edges from white cells towards non-white cells.

$$\bigvee_{d=0}^{3} R_{x,d} \tag{24}$$

$$\begin{aligned}\neg(R_{x,0} \wedge R_{x,1}) \wedge \neg(R_{x,0} \wedge R_{x,2}) \wedge \neg(R_{x,0} \wedge R_{x,3}) \wedge \\ \neg(R_{x,1} \wedge R_{x,2}) \wedge \neg(R_{x,1} \wedge R_{x,3}) \wedge \neg(R_{x,2} \wedge R_{x,3})\end{aligned} \tag{25}$$

$$(\neg Z_x \wedge \neg C_x) \rightarrow [(Z_d \vee C_d) \rightarrow \neg R_{x,d}] \tag{26}$$

As in the classic method, we must ensure that every white cell has exactly one incoming edge. However, in the novel method, we require exactly one incoming edge coming from a white cell. In addition to this edge, other incoming edges are allowed from black or indicative cells. That is why we first make sure that each white cell x has at least one incoming edge, which is guaranteed by formula 27. The variable $R_{d,x}$ denotes an edge from the neighbour in direction d of cell x to the current cell x.

Additionally, we have to guarantee that exactly one of these incoming edges comes from a white cell, even though additional edges coming from black or indicative cells may exist. To ensure this, we require that for every unique pair of incoming edges, both source cells cannot be white. In other words, at least one of the edges of a pair must come from a black or indicative cell. This is done with formula 28. It states that if the current cell x is white and there are two incoming edges from direction $d1$ and $d2$, then the corresponding source cells cannot both be white, meaning that at least one of them must be black or an indicative cell.

$$\neg Z_x \wedge \neg C_x \rightarrow \bigvee_{d=0}^{3} R_{d,x} \tag{27}$$

$$\bigwedge_{d1=0}^{2} \bigwedge_{d2=d1+1}^{3} [(\neg Z_x \wedge \neg C_x \wedge R_{d1,x} \wedge R_{d2,x}) \rightarrow (Z_{d1} \vee C_{d1} \vee Z_{d2} \vee C_{d2})] \tag{28}$$

On its own, this formula also allows only incoming edges from black or indicative cells. However, in combination with formulas 24 and 25, this situation turns out to be impossible. These two other formulas enforce that every white cell must have exactly one outgoing edge. Suppose we have a white cell x with only incoming edges from indicative cells. If we follow the outgoing edges of the white cells, starting at the white cell x, then we can have a white cell that has two options for its outgoing edge. The first option is to point its outgoing edge to a white cell that already has an incoming edge from another white cell, but this violates formula 28. Since this option is not valid, we have to create an outgoing edge to a white cell that has no incoming edges from another white cell, which is the second option. This way, we will eventually end in cell x, resulting in an incoming edge from a white cell and closing the cycle. This example is illustrated in Figure 7. In this figure, the white cell x is marked with a cross, and the green edges indicate the possible outgoing edges. As a result, every white cell always has exactly one incoming edge from another white cell, and we only have to prevent two incoming edges from other white cells.

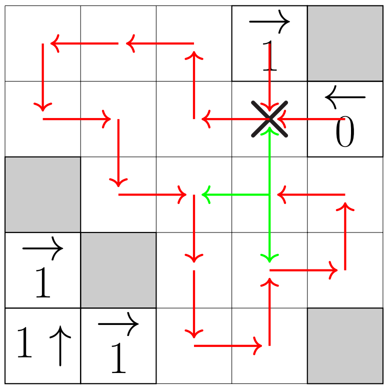

Figure 7: Example of how two incoming edges from indicative cells will result in a cycle.

To ensure that no cycles of length two occur in the cell tree, we have to prevent edges from pointing directly towards each other. This guarantees that there is always at least one corner of a cell inside a cycle. Formula 29 ensures this constraint. If there is an edge from cell x in direction d, then there cannot be an edge from its neighbour in direction d going back to cell x.

$$\bigwedge_{d=0}^{3} [R_{x,d} \rightarrow \neg R_{d,x}] \tag{29}$$

3.3.2 Corner tree

The corner tree covers the corners of the cells. It is not necessary to assign four variables to every corner of each cell on the board, as each corner is shared by at most four cells. The top left corner belongs to each cell $x = (i, j)$, as shown in Figure 8. The other three corners of the cell will belong to other cells. Each cell will be assigned four variables corresponding to its top left corner, one for each direction. The variable $H_{x,d}$ denotes the edge from the top left corner of cell x in direction d.

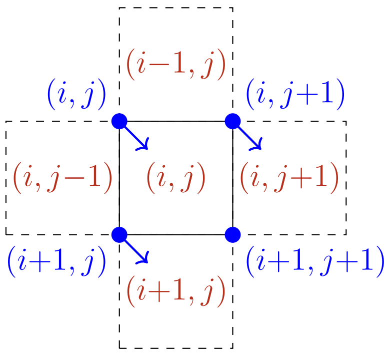

Figure 8: Mapping of the corners of cell (i, j).

This tree must also have exactly one outgoing edge per corner. This is ensured by formulas 30 and 31. These formulas use the same approach as we used to ensure exactly one outgoing edge in the cell tree. However, we do not require that each corner has exactly one incoming edge, as there may be multiple corner trees on the board. Each corner tree has a root corner with no incoming edges.

$$\bigvee_{d=0}^{3} H_{x,d} \tag{30}$$

$$\begin{aligned}&\neg(H_{x,0} \wedge H_{x,1}) \wedge \neg(H_{x,0} \wedge H_{x,2}) \wedge \neg(H_{x,0} \wedge H_{x,3}) \wedge \\ &\neg(H_{x,1} \wedge H_{x,2}) \wedge \neg(H_{x,1} \wedge H_{x,3}) \wedge \neg(H_{x,2} \wedge H_{x,3})\end{aligned} \tag{31}$$

What remains important is to prevent cycles within the corner tree. To avoid cycles of length two, we ensure that a cycle must consist of at least four corners. This is done in the same way as for the cell tree, ensuring that there is at least one cell inside a possible cycle in the corner tree. Formula 32 ensures that if there is an edge from the top left corner of cell x in direction d, then the edge in the opposite direction is not allowed. This prevents cycles between adjacent cells.

$$\bigwedge_{d=0}^{3} [H_{x,d} \to \neg H_{d,x}] \tag{32}$$

To ensure acyclicity in both trees, we have to verify that there are no crossing edges between the cell tree and the corner tree. We do this by handling each direction separately, as done with the formulas 33, 34, 35, 36. We use the numbers $0, 1, 2$ and 3 to denote the directions north, east, south, west, respectively. We explain only formula 33 here, as the other formulas follow a similar pattern, see Figure 9. If there is an edge from the top left corner of cell (i, j) going north, then there must be no edges in the cell tree between cell $(i-1, j-1)$ and cell $(i-1, j)$ in either direction.

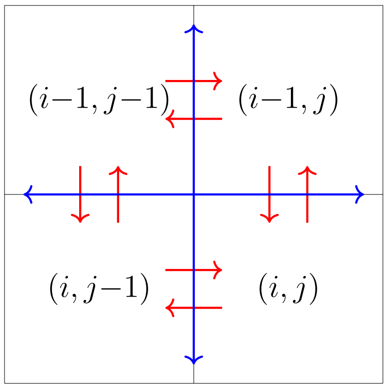

Figure 9: Logic of preventing crossing edges for each direction.

$$H_{h_{(i,j)},0} \to [\neg E_{x_{(i-1,j-1)},y_{(i-1,j)}} \wedge \neg E_{x_{(i-1,j)},y_{(i-1,j-1)}}] \tag{33}$$

$$H_{h_{(i,j)},1} \to [\neg E_{x_{(i-1,j)},y_{(i,j)}} \wedge \neg E_{x_{(i,j)},y_{(i-1,j)}}] \tag{34}$$

$$H_{h_{(i,j)},2} \to [\neg E_{x_{(i,j-1)},y_{(i,j)}} \wedge \neg E_{x_{(i,j)},y_{(i,j-1)}}] \tag{35}$$

$$H_{h_{(i,j)},3} \to [\neg E_{x_{(i,j-1)},y_{(i-1,j-1)}} \wedge \neg E_{x_{(i-1,j-1)},y_{(i,j-1)}}] \tag{36}$$

When these formulas are combined, and the starting edges are disregarded when we check for crossing edges, two different acyclic trees will be created [Rei98]. The cell tree will then contain the solution's cycle, when we disregard the edges coming from black and indicative cells.

3.4 CNF

The SAT solver only accepts formulas in CNF. Some of the propositional formulas discussed in the previous sections are not yet in CNF. If we want to convert a formula that uses an implication into CNF, such as $x \to y$, we have to rewrite this to $\neg x \vee y$. We can also have formulas such as $\neg(x \vee y)$. This formula can be rewritten using De Morgan's Law to $\neg x \wedge \neg y$. The OR operator is here changed to AND and vice versa.

As an example, we will convert formula 16 into CNF. First, we have to eliminate the implication, which results in formula 37. Second, we apply De Morgan's Law to remove the negation, which results in formula 38. Finally, by using the distributivity property of propositional formulas, we can transform the formula into the correct form. Formula 39 is the final formula. Similarly, we can transform all the previous formulas into CNF.

$$\neg(Z_d \vee C_d) \vee \neg E_{x,d} \tag{37}$$

$$(\neg Z_d \wedge \neg C_d) \vee \neg E_{x,d} \tag{38}$$

$$(\neg Z_d \vee \neg E_{x,d}) \wedge (\neg C_d \vee \neg E_{x,d}) \tag{39}$$

3.5 Variables

To be able to use all the previous formulas, we have to declare several variables. In this section, we will discuss the number of variables that we need, as well as the starting index of these variables. We will use n as the height of the board and m as the width.

First, we have to declare variables to represent black cells. These variables are denoted by Z_x, and their starting index is 1. Since every cell of the board needs such a variable, we need a total of $n \times m$ variables. We also need to keep track of indicative cells. We use the variable C_x to denote an indicative cell. Similarly, we need $n \times m$ variables, one for each cell. The starting index of the variables C_v is $(n \times m) + 1$ to prevent overlapping indices[1].

When we want to get the right number of black cells in a region, we have to count. To do this, we use the count variables $T_{v,t}$. How many of these variables we need depends on the configuration of the puzzle. It depends on how many indicative cells there are, in which position they are and the number inside the indicative cell. These variables are assigned to each cell within each region corresponding to an indicative cell, including the boundary cells. For this reason, we are not able to reserve a fixed number of variables. Therefore, we denote the required total of these variables that we need as $\#T$. The starting index for these variables is $(n \times m) + (n \times m) + 1$.

[1] All variables are numbered consecutively, as this is the required input format for the SAT solver.

So far, we have declared a total of $2 \times (n \times m) + \#T$ variables. The number of variables needed to construct the cycle depends on the chosen method. We will discuss the variables for each of these methods in the following sections.

3.5.1 Classic Method

When we choose to use the classic method, we declare the variable $E_{x,d}$ to represent directed edges between cells. We do not assign these variables to indicative cells because we know the positions of the indicative cells in advance, and these cells are not included in the cycle. The starting index for these edge variables is $2 \times (n \times m) + \#T + 1$. We assign four variables for each cell, one for each direction. These are in principle $4 \times (n \times m)$ variables in total. However, we do not assign an edge variable if it represents an edge towards the border of the board, resulting in $2 \times (n + m)$ variables that are not assigned. Since we do not assign edge variables to indicative cells, we subtract $\#I$ variables. This results in a total of $4 \times (n \times m) - 2 \times (n + m) - \#I$ variables that are needed to represent edges between cells.

To eventually construct the correct cycle, we have to declare count variables for each cell, except for indicative cells. These variables are denoted as $Q_{x,t}$. Each cell will get $n \times m$ variables, because in the worst case, the cycle passes through every other cell. In total, this results in $(n \times m)^2$ count variables for the whole board. However, indicative cells are excluded, so we only need $(n \times m)^2 - \#C$ variables, where $\#C$ is the number of indicative cells multiplied by $n \times m$. The starting index of these variables is $2 \times (n \times m) + \#T + 4 \times (n \times m) - 2 \times (n + m) - \#I + 1$. In the end, when using the classic method, we need $(n \times m)^2 + 6 \times (n \times m) + \#T - 2 \times (n + m) - \#I - \#C$ variables to get the correct solution.

3.5.2 Novel Method

The novel method uses two trees, each with its own set of variables. The cell tree uses variables $R_{x,d}$ to represent edges between cells. Every cell will get these variables, including the indicative cells (and black cells). The starting index of these variables is $2 \times (n \times m) + \#T + 1$. We have to assign $4 \times (n \times m)$ of these variables in total. However, variables corresponding to edges pointing towards the border of the board are excluded. Therefore, we only need $4 \times (n \times m) - 2 \times (n + m)$ variables to represent edges between cells.

In the corner tree, not every cell on the board will get the variable $H_{x,d}$. We exclude the cells in the first row and first column when assigning the variables. The reason is that we assign variables to the top left corner of each cell. The top left corner of the cells of the first row and column is on the border of the board, which is not a meaningful position for a corner edge. For our corner tree, we only need edges from the corners inside the board. Therefore, we will skip $n + m - 1$ cells when assigning variables. Every other cell will get four variables for its top left corner, resulting in a total of $4 \times (n \times m) - 4 \times (n + m - 1)$ variables. The starting index of these variables is $2 \times (n \times m) + \#T + 4 \times (n \times m) - 2 \times (n + m) + 1$.

In total, when using the novel method, the number of variables used is $10 \times (n \times m) - 4 \times (n + m - 1) + \#T - 2 \times (n + m)$. Therefore, the novel method requires fewer variables compared to the classic method. The main reason for this is that the number of variables grows quadratically in the classic method, while it is linearly increasing in the novel method when we increase the size of the puzzles.

3.6 Program

Each Yajilin puzzle is unique. That is why we need a program that declares the variables and makes the formulas for a specific puzzle. These variables and formulas are then passed to a SAT solver. Our implementation, written in C, first loads a puzzle from a .txt file, after which it generates the variables and formulas and passes these to the solver. Our program can distinguish between different types of cells, such as white cells, indicative cells and boundary cells. This makes it easier to skip over a certain type of cell when making the different variables and formulas. In the end, the SAT solver will try to determine if the problem is satisfiable and provide a solution if it is satisfiable.

We make use of the SAT solver Kissat 4.0.0 [Bie24], which has a built-in C library. This makes it easier to create the program. Kissat also provides some statistics, such as the runtime. The total number of variables needed is calculated by our program. These statistics are needed when we compare the efficiency of the classic and novel methods.

4 Experiments

In this chapter, we compare the efficiency of the classic and the novel method for solving Yajilin puzzles. For this, we make use of the uniquely solvable puzzles on the Janko website [JJb]. There are 590 puzzles in total available that are categorised by both size and difficulty. We analyse both categories to compare the two methods for size and difficulty. Our goal is to determine whether the novel method is always more efficient than the classic method, regardless of the size and difficulty of the puzzle.

4.1 Comparing the methods according to the size of the puzzle

We first have a look at the puzzles categorised by their size. The puzzles on the site are grouped in multiple sizes, but we also make our own categorisation, small, medium and large. Small puzzles have a size between 7×7 and approximately 200 cells (denoted as ~ 200). Medium puzzles range from 15×15 to ~ 500. Finally, large puzzles have a size between 25×25 and ~ 1000 or more $(+)$. The symbol '+' is used on the website to indicate puzzles of an extremely large size.

For each category, we test both methods on 20 puzzles. We only use 20 puzzles to maintain a fair distribution of sizes within each group, because there are sometimes not enough puzzles available of a specific size. The puzzles that are used are selected randomly. For every puzzle, we analyse both the runtime and the total number of variables used to solve the puzzle.

Table 2 presents the results of both methods tested on small puzzles, Table 3 and Table 4 show the results on medium and large puzzles. The numbers in the column 'Puzzle' refer to the puzzles on the Janko website [JJb]. What immediately stands out is that both methods are very fast in solving small and medium puzzles, while the novel method is slightly faster on average and in all but 8 cases. The average runtime of medium puzzles, when using the novel method, is affected by puzzle 384. When disregarding this value, the novel method is a lot faster in solving medium-sized puzzles. Additionally, the novel method requires fewer variables compared to the classic method. This is mainly because the classic method roughly requires $(n \times m)^2$ variables to construct the loop, whereas the novel method only uses a linear number of variables for the two trees.

		Classic Method		*Novel Method*	
Size	**Puzzle**	**Runtime (s)**	**Variables**	**Runtime (s)**	**Variables**
7×7	1	0.08	2402	0.03	462
7×7	41	0.08	2353	0.02	515
7×7	332	0.07	2540	0.04	490
10×10	35	0.42	9312	0.03	1090
10×10	54	0.17	9287	0.03	1064
10×10	226	0.17	10115	0.05	1063
10×10	537	0.19	8275	0.03	988
11×11	564	0.08	12936	0.04	1440
11×11	143	0.13	14205	0.2	1339
12×12	144	0.14	19873	0.17	1685
12×12	555	0.18	19548	0.05	1514
13×13	543	0.15	25705	0.04	2209
13×13	147	0.41	27179	0.06	2131
14×14	479	0.17	32290	0.04	2344
14×14	281	0.47	35147	0.06	2204
~100	98	0.17	7932	0.04	964
~200	231	0.22	32006	0.07	2067
~200	314	0.47	25889	0.44	1758
~200	523	1.25	37060	1.89	2434
~200	201	0.22	30152	0.07	1863
Average		**0.262**	**18210**	**0.17**	**1481**

Table 2: Results of small-sized puzzles for the classic and novel method.

		Classic Method		*Novel Method*	
Size	**Puzzle**	**Runtime (s)**	**Variables**	**Runtime (s)**	**Variables**
15×15	415	0.23	42628	0.06	4008
15×15	519	0.24	41331	0.06	3856
15×15	521	0.47	41661	0.07	3041
16×16	549	0.36	54941	0.09	3821
17×17	169	0.91	76792	0.22	3375
17×17	561	0.29	62013	0.07	5593
18×10	37	0.18	28614	0.04	1978
18×10	119	0.55	30739	0.2	2256
$\sim$250	233	0.53	39196	0.8	2234
$\sim$250	284	0.66	40982	0.67	2277
$\sim$300	384	1.57	76453	32.89	3577
$\sim$300	264	0.84	60052	0.2	2802
$\sim$300	524	0.93	75681	1.44	3551
$\sim$350	7	2.09	103625	0.94	3820
$\sim$350	250	1.33	105850	1.33	4011
$\sim$350	492	1.13	83639	6.29	4004
$\sim$400	268	2.96	128045	0.57	4254
$\sim$400	545	2.57	102400	0.29	3964
$\sim$500	290	47.25	184531	4.86	5906
$\sim$500	579	1.59	150005	0.11	7645
Average		**3.334**	**76459**	**2.56**	**3799**

Table 3: Results of medium-sized puzzles for the classic and novel method.

		Classic Method		*Novel Method*	
Size	**Puzzle**	**Runtime (s)**	**Variables**	**Runtime (s)**	**Variables**
25×25	140	10.35	309220	1.01	10214
30×30	527	26.05	673462	4.68	16884
36×20	20	12.86	464069	3.81	12759
36×20	39	19.49	463189	16.58	8988
36×20	10	103.19	467163	117.85	9338
45×31	30	126.31	1706514	N/A	20381
45×31	40	218.17	1718292	20.68	20968
$\sim$600	385	10.25	250224	5.58	6872
$\sim$600	460	9.15	275956	2.61	9609
$\sim$600	510	8.92	238013	642.46	6696
$\sim$700	557	9.04	363483	8.3	11603
$\sim$700	587	16.35	410746	1.28	9023
$\sim$800	420	25.87	490058	1964.42	11014
$\sim$800	560	16.28	471751	90.03	11764
$\sim$900	562	16.87	629282	1.72	15321
$\sim$900	430	19.27	582682	1.89	12064
$\sim$1000	589	16.93	677907	0.81	26993
$\sim$1000	578	40.76	898795	196	15986
+	528	583.84	4121767	N/A	41485
+	572	52.48	1441017	1.13	41835
Average		**67.122**	**832680**	**171.158**	**15990**

Table 4: Results of large-sized puzzles for the classic and novel method.

It is important to note that some puzzles take significantly longer to solve when using the novel method, especially when compared to the classic method. These puzzles with relatively large runtimes are, most of the time, large puzzle. Examples are puzzles $384, 510, 420, 560$ and 578. Furthermore, puzzles 384 and 420 are also puzzles that take longer to solve in comparison with other puzzles when we use the classic method. We do not have a well-founded reason why these puzzles take a long time to solve. We will discuss these puzzles in more detail in section 4.3.

We also observe that many puzzles are solved in a short time when using the novel method compared to using the classic method. Examples are puzzles $7, 290$ and several large puzzles, such as 572 and 589. The reason that the first method takes a long time in this case is probably because the number of variables is extremely high. When the puzzle size increases, the number of variables is a lot higher for the classic method compared to the novel method. This difference likely explains why the novel method is, on average, better at solving these puzzles.

However, there are two large puzzles, puzzles 30 and 528, for which the novel method was unable to find a solution within a reasonable time. These puzzles are also harder to solve for the classic method in comparison with other puzzles. We ran the program for several hours on these puzzles, but the program did not terminate when we used the novel method. We try to find a reason for this behaviour in section 4.3.

4.2 Comparing the methods according to the difficulty of the puzzle

We also have a look at the difference in efficiency between the two methods for different difficulty levels. We will use the categorisation of the Janko website [JJb]. This difficulty level is mainly based on the time required for a human to solve the puzzle [JJa]. There are nine difficulty levels, 1 to 8 and 'schwer'. For each difficulty level, we test both methods on six puzzles, so 54 puzzles in total. We mainly reuse the puzzles from previous tests, but these puzzles do not have a fair distribution of difficulty levels. That is why we have to remove some of these puzzles and add new ones to other difficulty levels. Again, all the puzzles are chosen randomly.

Table 5 presents the results per difficulty level for both the classic and novel method. We see that the runtime increases as the difficulty level increases, as expected. In general, the higher the difficulty, the longer it takes for both methods to find a solution. The runtime increases steadily with the classic method, whereas the novel method shows some outliers. The total number of variables is also increasing when the puzzles get more difficult. Again, we see that the novel method is faster most of the time in solving the puzzles and uses fewer variables, regardless of the difficulty level. Furthermore, the runtime for the novel method appears to increase with higher difficulty levels, although there are exceptions. Exceptions occur between levels 4 and 5, and levels 7 and 8, due to single outliers in difficulty levels 4 and 7. Most of the time, a high difficulty level indicates that the puzzle is bigger or that there are relatively fewer hints. Puzzles with a low difficulty level are solved more efficiently using the novel method. Although these puzzles are relatively small, the novel method is still 3 to 6 times faster and requires 5 to 12 times fewer variables.

		Classic Method		*Novel Method*	
Difficulty	**Puzzle**	**Runtime (s)**	**Variables**	**Runtime (s)**	**Variables**
1	1	0.08	2402	0.03	462
	41	0.08	2353	0.02	515
	537	0.19	8275	0.03	988
	555	0.18	19548	0.05	1514
	98	0.17	7932	0.04	964
	403	0.24	9491	0.05	1063
Average		***0.157***	***8334***	***0.037***	***918***
2	35	0.42	9312	0.03	1090
	54	0.17	9287	0.03	1064
	479	0.17	32290	0.04	2344
	201	0.22	30152	0.07	1863
	413	0.34	9515	0.04	1087
	94	0.41	9635	0.04	1099
Average		***0.288***	***16699***	***0.042***	***1425***
3	332	0.07	2540	0.04	490
	314	0.47	25889	0.44	1758
	561	0.29	62013	0.07	5593
	37	0.18	28614	0.04	1978
	233	0.53	39196	0.8	2234
	264	0.84	60052	0.2	2802
Average		***0.397***	***36384***	***0.265***	***2476***
4	564	0.08	12936	0.04	1440
	543	0.15	25705	0.04	2209
	523	1.25	37060	1.89	2434
	521	0.47	41661	0.07	3041
	384	1.57	76453	32.89	3577
	250	1.33	105850	1.33	4011
Average		***0.808***	***49944***	***6.043***	***2785***
5	143	0.13	14205	0.2	1339
	147	0.41	27179	0.06	2131
	119	0.55	30739	0.2	2256
	7	2.09	103625	0.94	3820
	492	1.13	83639	6.29	4004
	545	2.57	102400	0.29	3964
Average		***1.147***	***60298***	***1.33***	***2919***
6	549	0.36	54941	0.09	3821
	524	0.93	75681	1.44	3551
	290	47.25	184531	4.86	5906
	579	1.59	150005	0.11	7645
	39	19.49	463189	16.58	8988
	587	16.35	410746	1.28	9023
Average		***14.328***	***223182***	***4.06***	***6489***

	385	10.25	250224	5.58	6872
	510	8.92	238013	642.46	6696
7	470	1.43	78700	2.3	3528
	428	2.47	82190	0.67	4182
	247	5.37	105129	3.16	4305
	18	2.78	99362	0.85	5330
Average		***5.203***	***142270***	***109.17***	***5152***
	527	26.05	673462	4.68	16884
	20	12.86	464069	3.81	12759
8	10	103.19	467163	117.85	9338
	460	9.15	275956	2.61	9609
	562	16.87	629282	1.72	15321
	589	16.93	677907	0.81	26993
Average		***30.842***	***531307***	***21.913***	***15151***
	140	10.35	309220	1.01	10214
	30	126.31	1706514	N/A	20381
Schwer	557	9.04	363483	8.3	11603
	420	25.87	490058	1964.42	11014
	560	16.28	471751	90.03	11764
	528	583.84	4121767	N/A	41485
Average		***128.615***	***1243799***	***343.96***	***17744***

Table 5: Results of the puzzles, grouped per difficulty level, for both the classic method and novel method.

4.3 Interesting Results

Since there are several puzzles with results that we did not expect, we tried to find a reason why these puzzles are hard to solve. In some cases, one method was slower than the other. However, there are also puzzles which took longer to solve for both methods. That is why we looked at the structure of the puzzle rather than the methods themselves. First, we looked at the total number of black and indicative cells. It might be the case that one of the methods struggles in finding the white cells which are included in the cycle when there are fewer black and indicative cells, resulting in more white cells. However, when comparing the total number of black and indicative cells to other puzzles, we could not find a consistent pattern; specifically, the runtime does not increase when there are fewer black and indicative cells. We also tried to find a pattern in the total number of black cells that are not indicated by any indicative cell, since the methods need to find these cells based on the construction of the cycle. But again, there was no consistent pattern to be found; the runtime does not increase when there are more such black cells.

It might be the case that the distribution of these black and indicative cells plays a big role in the difference in runtime. We observed that some puzzles that were very hard to solve with the novel method contained several large white areas in the initial state of the puzzle. This might increase the complexity of finding a solution, since there are more cells next to each other that might contain the cycle. It turned out to be quite difficult to compare the distribution of white areas in a meaningful way to other puzzles. So this remains an open question for further research.

5 Conclusions and Further Research

Yajilin is a single-player logic puzzle. The aim is to place black cells to create a cycle that only passes through white cells. The indicative cells give you hints where to place black cells. To solve these puzzles, we encode the rules of the puzzle into propositional formulas and use a SAT solver to find the solution. We are interested in whether the novel method from Klaus Reinhardt [Rei98] is more efficient than the classic method used by Gerhard van der Knijf [vdK21].

The classic method makes use of counting the distance from each cell to a starting cell. This method needs $n \times m$ variables per cell. As a result, the total number of variables is increasing rapidly as the size of the puzzle increases.

The novel method, introduced by Klaus Reinhardt [Rei98], constructs two trees. The first tree covers the cells of the puzzle. The second tree covers the corners of these cells. These two trees together prevent the existence of a cycle. By disregarding an edge when checking for crossing edges between the two trees, we can get a solution with a cycle. For each tree, we need a constant number of variables per cell, regardless of the size of the puzzle.

To evaluate the efficiency of these two methods, we made a program in C. This program reads Yajilin puzzles from the Janko website [JJb]. After this, it declares all variables that are needed and passes them to the SAT solver, which tries to find a solution.

The results suggest that the novel method is most of the time more efficient in solving the puzzles. However, some puzzles are very hard to solve with the novel method, although they are sometimes easy to solve with the classic method. It is still unclear why this happens. The results also show that if the puzzle size or difficulty level increases, the more time and variables it takes to solve the puzzle, regardless of the method that is used. Furthermore, the number of variables used in the classic method increases faster compared to the novel method.

There are several interesting topics to consider for future research. First, we determine the starting cell and starting edges manually. To do this, we need to know some part of the solution in advance. It is interesting to see if the novel method is still more efficient when the program searches for the starting cell or edges. Second, it might be worth optimising both methods. Currently, we do not optimise the number of variables that are used. We can save a lot of variables in the classic method. For example, we can use fewer variables per cell to represent the number of the cell by considering the minimal number of black cells. Finally, it is important to understand why certain puzzles are extremely hard to solve with the novel method. If we know the reason, we might choose the classic method over the novel method for certain puzzles.

References

[AAW24] Muhammad Tsaqif Ammar, Muhammad Arzaki, and Gia Septiana Wulandari. Efficient SAT-Based Approach for Solving Juosan Puzzles. In Dieky Adzkiya and Kistosil Fahim, editors, *Applied and Computational Mathematics*, pages 211–226, Singapore, 2024. Springer Nature Singapore.

[Bie24] Armin Biere. Kissat. https://github.com/arminbiere/kissat, 2024.

[Hes24] Niels Heslenfeld. Solving and generating Fobidoshi puzzles. Bachelor thesis informatica, LIACS, Leiden University, 2024.

[Hoo] Hendrik Jan Hoogeboom. SAT Encoding Connectedness in a Planar Grid. Unpublished.

[ISI12] Ayaka Ishibashi, Yuichi Sato, and Shigeki Iwata. NP-completeness of Two Pencil Puzzles: Yajilin and Country Road. *Utilitas Mathematica*, 88, 07 2012.

[JJa] Angela Janko and Otto Janko. Rätsel, Puzzles & Denksport online. https://www.janko.at/Raetsel/index.htm#schwierigkeit. Accessed 20-05-2025.

[JJb] Angela Janko and Otto Janko. Yajilin. https://www.janko.at/Raetsel/Yajilin/index.htm. Accessed 17-02-2025.

[Nik] Nikoli. Yajilin. https://www.nikoli.co.jp/en/puzzles/yajilin/. Accessed 20-02-2025.

[Rei98] Klaus Reinhardt. On some recognizable picture-languages. In Luboš Brim, Jozef Gruska, and Jiří Zlatuška, editors, *Mathematical Foundations of Computer Science 1998*, pages 760–770, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[Str25] Hanna Straathof. Solving and generating Kuroshuto puzzles. Bachelor thesis informatica, LIACS, Leiden University, 2025.

[vdK21] Gerhard van der Knijff. Solving and generating puzzles with a connectivity constraint. Bachelor thesis computing science, Radboud University, 2021.

[Wen24] Roos Wensveen. Solving, Generating and Classifying Hitor. Master computer science, LIACS, Leiden University, 2024.