J. Indones. Math. Soc.
Vol. 32, No. 02 (2026), pp. 1–27.

A SAT-Based Approach for Solving Suguru Puzzles: Theory and Experiment

Butrahandisya[1] and Muhammad Arzaki[1,2*]

[1] Computing Laboratory, School of Computing, Telkom University, Indonesia
[2]Faculty of Information Technology, Monash University, Australia

Abstract. We discuss a SAT-based approach for solving Suguru puzzles—one-player puzzles similar to Sudoku that were confirmed NP-complete in 2022. We first discuss the formal rules of the puzzles and provide a rigorous technique to translate such rules into propositional formulas in conjunctive normal form (CNF). The resulting formulas form what is called a SAT encoding, and we prove that the number of clauses and variables in our encoding is polynomially proportional to the puzzle's dimension. This encoding allows one to reduce Suguru puzzles to SAT problems, and we use this encoding to construct a declarative SAT-based program without any search algorithm in C++ for solving general Suguru puzzles. We perform experiments involving 230 test cases for Suguru instances of size $n \times n$ where $6 \leq n \leq 15$. Finally, we derive some empirical results from these experiments and argue that, in terms of running time, our SAT-based approach outperforms the previously proposed backtracking technique in solving larger puzzles.

Key words and Phrases: SAT encoding, SAT problems, SAT solver, Suguru puzzles.

1. Introduction

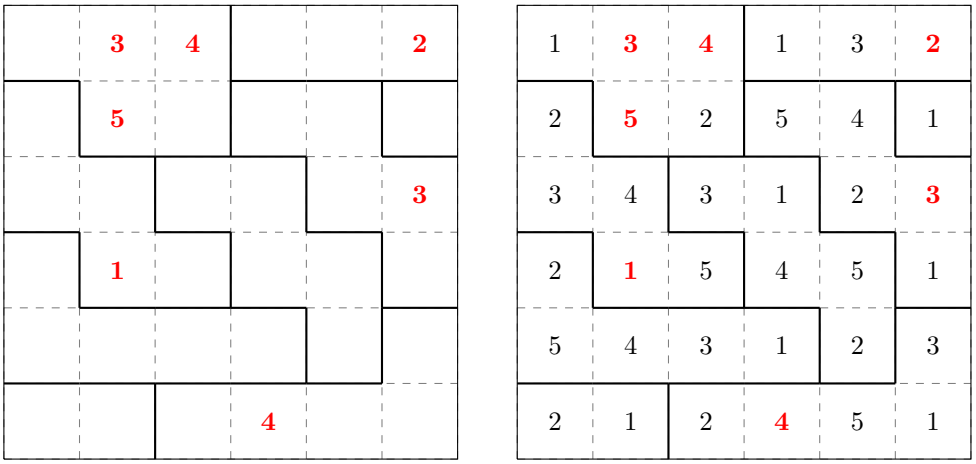
Single-player games that involve filling rectangular grids with numbers or symbols according to specific rules can be both challenging and captivating. Some of these games are inherently challenging and have a similar interesting property: finding solutions to these puzzles is typically "hard", but once the solutions are found, checking whether these solutions satisfy the puzzles' rules is relatively "easy".[1] A

*Corresponding author: arzaki@telkomuniversity.ac.id, muhammad.arzaki@monash.edu
2020 Mathematics Subject Classification: 03B70, 05-04, 68T27, 68W30
Received: 10-02-2025, accepted: 25-11-2025.

[1]We refrain from formally defining the notion of "hard" and "easy" in this introduction, but "hard" informally refers to the condition in which no polynomial time algorithm is known to solve such a problem in general settings. "Easy" means that the solution to such a problem can be found in polynomial time.

famous example of these puzzles is Sudoku. Another example is Suguru, a one-player game invented by Naoki Inaba, a prominent Japanese logic puzzle designer.

Suguru first emerged in 2001 [1] and gained popularity in the theoretical computer science community in 2022 after Robert et al. proved its NP-completeness [2]. The puzzles are played in rectangular grids; a grid is partitioned into one or more *regions*. Each region contains one or more orthogonally connected cells. Initially, some cells in the grid might be empty or filled with positive integers. These pre-filled cells are called the *hint cells* or simply *hints*. The objective of this puzzle is similar to the famous Sudoku—each empty cell must be filled with exactly one positive integer satisfying particular constraints. Suguru puzzles basically only have two rules: (1) for every region, each cell in that region must be filled with a unique integer between one and the number of cells in such a region, and (2) any two adjacent cells in the grid, either orthogonally or diagonally, must be filled with distinct integers.



(A) A Suguru puzzle instance of size 6×6.

(B) A solution to the instance in Fig. 1a.

FIGURE 1. An instance of a Suguru puzzle (Fig. 1a) and its solution (Fig. 1b). The bold numbers in the cells indicate the hints of the puzzle (the pre-filled cells). Notice that the solution in Fig. 1b satisfies all rules of Suguru puzzles.

Fig. 1 illustrates an example of a Suguru puzzle instance and its solution. Here, we consider a Suguru puzzle of size 6×6. The instance in Fig. 1a consists of nine regions: there are six regions of size 5 and one region each of size 1, 2, and 3. The instance has seven hint cells. Notice that the solution to this instance in Fig. 1b preserves these hint cells.

The computational aspects of various one-player puzzles have attracted interest in the theoretical computer science community, especially in their connection

to NP-complete problems. In the last six years, many single-player logic puzzles were proven NP-complete, such as Calculation Solitaire [3], Choco Banana [4], Juosan [5], Path Puzzles [6], Suguru [2], Tatamibari [7], Tilepaint [8], Yin-Yang [9], and ZHED [10]. Various techniques have been proposed to solve some of these NP-complete puzzles. Most of them consider imperative approaches, such as the exhaustive search for solving Tatamibari [11], the prune-and-search technique for solving Yin-Yang [12], as well as various backtracking approaches for solving Juosan [13], Path Puzzles [14], Suguru [15], and Tilepaint [16]. Typically, these algorithms involve sophisticated search-based criteria in their computational steps. One of them, namely the backtracking approach for solving Suguru puzzles explained in [15], exhibits a factorial upper bound for its running time complexity—which is not optimal for an NP-complete problem.

The Boolean satisfiability problem (SAT problem) is a fundamental problem in theoretical computer science and mathematical logic. It is the first established NP-complete problem [17], [18], [19], [20]. Theoretically, all NP-complete problems can be transformed into SAT problem instances. This property enables various NP-complete problems to be solved by SAT solvers—algorithms specifically designed to determine the satisfiability of propositional formulas. Due to its versatility, SAT solvers have been used to assist various problems in computer science, such as automated planning and scheduling [21], automated theorem proving [22], formal software or hardware verification [23], [24], solving complex optimization problem [25], and various aspects of artificial intelligence [26].

Numerous studies have investigated SAT-based approaches for solving single-player logic puzzles, such as binary puzzles [27], edge matching puzzles [28], fill-a-pix [29], Juosan [30], Path Puzzles [31], Skyscraper [32], Sudoku [33], [34], [35], and other various single player puzzles [36]. A recent extensive investigation by Bright et al. demonstrates the effectiveness of the SAT-based approach in solving puzzles using the *declarative* approach [36]. In this approach, one focuses on describing the problems and their constraints as sets of Boolean formulas, rather than providing a sequence of imperative commands. Some empirical performance comparisons of the SAT-based and imperative search-based approaches for solving logic puzzles were recently discussed in [30], [31]. Ammar et al. showed that Juosan puzzles with up to 1350 cells can be solved in less than one second on a standard personal computer, significantly outperforming the backtracking approach, which required four days or more to solve the same puzzles on the same machine [30]. Sakti et al. showed that the SAT-based approach outperforms the backtracking algorithm for solving Path Puzzles if the number of cells is sufficiently large [31]. Nevertheless, one should note that, in earlier studies, the imperative paradigms still outperformed the SAT-based ones for small-sized puzzles.

This paper comprehensively discusses the SAT-based approach for solving Suguru puzzles. The analysis of this approach is discussed rigorously from both theoretical and experimental perspectives. The rest of the section in this paper is divided into four main parts. Section 2 discusses the preliminaries and related

works, including an outline of the earlier algorithmic investigation of Suguru puzzles and some notational conventions. Section 3 comprehensively explains Suguru puzzles as SAT problem instances, which includes the SAT encoding for Suguru puzzles and its analysis related to the number of variables and clauses of the resulting propositional formulas. Section 4 presents the experimental results and their analyses, including the running time comparison between the previous backtracking technique and our proposed SAT-based approach. Finally, Section 5 concludes the paper with important findings and some open problems.

2. Preliminaries and Related Works

2.1. Representation and Rules of Suguru Puzzles

An instance of a Suguru puzzle can be represented using a two-dimensional grid, divided into one or more regions, as depicted in Fig. 1. The objective of this puzzle is to fill each cell with a number satisfying the following constraints:

(1) every two distinct cells in a region must contain two different integers;
(2) the integer in a cell within a region must be between one and the number of cells in such a region;
(3) two adjacent cells in the grid, either orthogonally or diagonally, must contain two different integers.

Notice that the first two constraints above are derived from the first rule stating that each cell in a region must be filled with a unique integer between one and the number of cells in the region it belongs to. We divide the first rule into two constraints to make the translation of the rules into Boolean formulas more understandable.

A natural way to represent a Suguru instance and its solution is by using two-dimensional arrays as explained in [15]. However, we refrain from discussing this data structure in detail because it is irrelevant to our SAT-based approach for solving this puzzle. Nevertheless, we introduce some notations similar to those in [15] to ease our analysis, namely:

(1) the size of the grid is $m \times n$ if it contains m rows and n columns,
(2) a cell in an $m \times n$ grid is denoted by (i, j) where $1 \leq i \leq m$ and $1 \leq j \leq n$,
(3) the overall number of regions in the grid is denoted by R,
(4) the overall number of hint cells in the grid is denoted by H,
(5) a region is labeled with an integer k ($1 \leq k \leq R$); for a region k, the number of cells in this region is denoted by s_k.

Other notations are introduced when we discuss the representation of a Suguru puzzle instance as a SAT problem.

The cells are numbered in row-major order, i.e., the top-leftmost cell is the cell $(1, 1)$ and the bottom-rightmost cell is the cell (m, n). The regions are also numbered in a row-major order, with the first visited region being region 1 while the last visited region being region R. For example, in Fig. 1, the top-leftmost cell is the cell $(1, 1)$ and the bottom-rightmost cell is the cell $(6, 6)$. Moreover, in Fig. 1, there are nine regions (thus $R = 9$) with $s_1 = 5$ (the region with three hint cells

containing 3, 4, and 5), $s_2 = 3$ (the region with one hint cell containing 2), $s_3 = 5$ (the region with one hint cell containing 1), $s_4 = 5$ (the region with one hint cell containing 3), $s_5 = 1$ (the region containing a single cell), $s_6 = s_7 = 5$, $s_8 = 2$, and $s_9 = 5$ (the region with one hint cell containing 4). In Fig. 1, we also have $H = 7$.

2.2. **Recent Algorithmic Investigation of Suguru Puzzles**

Some general forms of Suguru puzzles were recently proven NP-complete in 2022 by Robert et al. [2]. The proofs involve a polynomial-time reduction from the Planar Circuit SAT problem, known to be NP-complete, to the Suguru puzzle. However, Robert et al. did not explicitly discuss the algorithm for solving the Suguru instance in general.

Recently, Butrahandisya et al. have discussed elementary algorithmic approaches for solving Suguru puzzles [15]. They proved that verifying whether a configuration is a solution to a Suguru instance takes polynomial time in terms of the puzzle's size. Specifically, this verification takes $O(mn)$ time for a Suguru configuration of size $m \times n$. They also proposed an optimized backtracking algorithm for solving arbitrary $m \times n$ Suguru instance with R regions and H hints cells in $O(R \cdot (mn - H + 2)!)$ time. This result provides a factorial upper bound for solving Suguru puzzles in general, although it is not the optimal one since Suguru puzzles are NP-complete. Their experimental results showed that Suguru instances with no more than 100 cells can be solved using a personal computer in less than 0.5 seconds. On the theoretical side, they also proved that any Suguru instance of size $m \times 1$ or $1 \times n$ is tractable (i.e., solvable in polynomial time) for arbitrary m and n. This result provides a special subproblem of Suguru puzzles where the solution can be found quickly.

Despite the positive results introduced in [15], the backtracking approach for solving the Suguru puzzle still has some limitations. Theoretically, the asymptotic upper bound for this algorithm is still factorial in terms of puzzle size, which makes it not optimal (in terms of the running time for an NP-complete problem) and might make it practically less efficient for larger puzzles. In addition, the algorithm is not straightforward to implement due to its complex search criteria (cf. [15, Algorithm 2]). This complicated search criteria also makes the algorithm struggle to solve specific types of instances in which it encounters many invalid states during its exploration before finally settling on the correct solution states. The general idea of the optimized backtracking algorithm for solving Suguru puzzles in [15] is as follows:

(1) Suppose we are given an instance of a Suguru puzzle. The cells are filled in row-major order. The hint cells are left unchanged. For each empty cell (i, j), the algorithm attempts to fill it with a value from a possible set of integers corresponding to the region to which the cell (i, j) belongs. In particular, if a cell (i, j) belongs to the region k, then the algorithm tries to fill (i, j) with an integer from the set $\{1, 2, \ldots, s_k\}$ (in increasing order).

(2) Suppose the algorithm assigns an integer t to a cell (i, j), then it checks whether a *backtracking* is necessary based on the following conditions:
   (a) if any of the cells adjacent to (i, j) is filled with an integer t, then the algorithm performs a backtrack;
   (b) if there is another cell within the same region as the cell (i, j) that is filled with an integer t, then the algorithm also performs a backtrack.

   A backtrack is performed by undoing the filling of the cell (i, j) and then trying to fill (i, j) with another integer. However, if there are no values that can fill the cell (i, j), the backtracking is performed further by moving back to the previous cell (in row-major order), and changing the entry of this previous cell with another integer.
(3) The algorithm terminates if exactly one of the following conditions is reached:
   (a) all cells have been successfully filled, here the algorithm terminates, and a solution to the instance is found;
   (b) the algorithm backtracks to the cell $(1, 1)$ (i.e., the top-leftmost cell), here the algorithm terminates and concludes that the instance has no solution.

2.3. **Propositional Logic, Conjunctive Normal Form (CNF), Satisfiability (SAT) Problem, and SAT Solver**

Propositional logic is a simple but powerful logical system that serves as the building block for other logical systems [37]. This system is constructed using *propositions* and logical *connectives* (or *operators*). A proposition is a statement with a truth value, either true or false (but not both). These propositions are represented using Boolean variables that can be combined using logical operators to form formulas.

In this paper, we assume that the readers are familiar with the syntax and semantics of propositional logic. For extensive references, see, e.g., [37, Chapter 2] and [38, Chapter 1]. Here, we only discuss some concepts and notations that are relevant to our approach. The symbols $\neg$, $\wedge$, and $\vee$ respectively denote negation, conjunction, and disjunction operators.

An *atomic proposition* is a proposition that cannot be further decomposed. A *literal* is either an atomic proposition or its negation. A *clause* is a disjunction of one or more literals. A formula is in *conjunctive normal form* (CNF) if it is a conjunction of one or more clauses. More formally, a formula ϕ is in CNF if it is defined by the following Backus-Naur Form grammar:

$$\begin{aligned} \lambda &::= p \mid \neg p \\ \alpha &::= \lambda \mid \lambda \vee \alpha \\ \phi &::= \alpha \mid \phi \wedge \alpha. \end{aligned} \tag{1}$$

In (1), λ denotes a literal, α denotes a clause, and ϕ denotes a formula in CNF. According to [37, Theorem 4.3] and [38, Section 1.5.2], every propositional logic formula can be transformed into an equivalent CNF formula. In this paper, we

consider CNF as the canonical form for propositional formulas (instead of other forms, such as the *disjunctive normal form* (DNF)) because it is a standard input form for many SAT solvers.

The SAT problem, also known as the Boolean satisfiability problem, is a problem of determining the *satisfiability* of a propositional logic formula, i.e., finding out if there exists an interpretation (a well-defined truth value assignment for each of the propositional atoms) that makes the formula true [18], [20], [37], [38]. This problem is the first established NP-complete problem (see, e.g., [17], [18, Lemma 34.5 and Lemma 34.6], [19], and [20, Theorem 7.37]). Technically, the NP-completeness of the SAT problem implies that every NP-complete problem is reducible to it. Moreover, since NP-complete problems are reducible to one another, every NP-complete problem is also reducible to the SAT problem.

Currently, the NP-completeness of the SAT problem intuitively means that checking whether an interpretation makes a propositional formula true is easy (i.e., it can be done in polynomial time), but finding such an interpretation is not. This intuition also applies to other NP-complete problems. For Suguru puzzles, checking whether a configuration is a solution to a Suguru instance is easy, while finding such a solution from scratch is generally hard.[2]

A SAT solver, as the name suggests, is a computer program specifically designed for solving the SAT problem. Since all NP-complete problems are reducible to the SAT problem, these problems can be transformed into the SAT problem. A SAT solver takes a propositional formula as an input, typically in CNF, and outputs the satisfiability status of such a formula (i.e., whether the formula is satisfiable or not). It has been extensively studied in computer science (see, e.g., [37, Chapter 6] for extensive preliminary references). When this paper is compiled, the fastest algorithm known for solving the SAT problem is the biased-PPSZ algorithm discussed by Hansen et al. in [39]. Until now, the optimal upper bound for the asymptotic complexity of algorithms for solving the SAT problem remains an open question.

The NP-completeness proof of the Suguru puzzle reduces the Planar Circuit SAT problem, a type of SAT problem, to the Suguru puzzle. However, it is natural to ask whether the explicit reverse reduction from the Suguru puzzle to the SAT problem is also possible. This paper partially addresses this concern by discussing an efficient (i.e., polynomial-size) encoding from the Suguru puzzle to the SAT problem. This encoding allows the puzzle to be solved using a SAT solver efficiently, as in the case of other NP-complete one-player games (see, e.g., [27], [30], [31], [32]).

3. Suguru Puzzles as SAT Problem Instances

This section discusses a systematic approach for encoding Suguru puzzle instances and rules into propositional formulas. Suppose we consider an instance represented using an $m \times n$ grid divided into R regions. For a region k ($1 \leq k \leq R$),

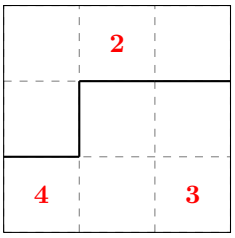
[2]Recall that, informally, we call a computational problem *hard* if there is no known polynomial time algorithm to solve such a problem.

s_k denotes the size of such a region (i.e., the number of cells in such a region). We define $r_{i,j} = k$ if and only if the cell (i, j) belongs to a region k. We can say that $r_{i,j}$ is the region label for the cell (i, j) in a Suguru instance grid.

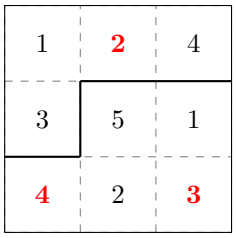
To represent a Suguru puzzle as a SAT formula, we define a propositional variable $g_{i,j,v}$ that is true if and only if the cell (i, j) contains an integer v. Here, $1 \leq i \leq m$, $1 \leq j \leq n$, and $1 \leq v \leq s_k$ where $r_{i,j} = k$. The definition of this propositional variable is similar to that used in the Sudoku puzzle (see, e.g., [33], [34], and [40, Section 1.3]). We also write $g_{i,j,v} = 1$ to signify that $g_{i,j,v}$ is true and $g_{i,j,v} = 0$ to signify that $g_{i,j,v}$ is false (which is equivalent to $\neg g_{i,j,v}$ is true). In practice, we can also define $g_{i,j,v} = 0$ for a cell (i, j) with $r_{i,j} = k$ if $v > s_k$.

3.1. **SAT Encoding for Suguru Puzzles**

A non-empty Suguru instance is represented as a propositional formula. In particular, the instance corresponds to a conjunction of several atomic propositions of the form $g_{i,j,v}$ where (i, j) is the hint cell filled with value v.



(A) An instance of a Suguru puzzle of size 3×3.



(B) A solution to the instance in Fig. 2a.

FIGURE 2. An example of a Suguru instance (Fig. 2a) and its solution (Fig. 2b). Bold numbers in the cells indicate hint cells. The solution in Fig. 2b fulfills the aforementioned rules of the Suguru puzzle.

For example, suppose we consider a 3×3 Suguru puzzle in Fig. 2. The instance in Fig. 2a corresponds to the formula $g_{1,2,2} \wedge g_{3,1,4} \wedge g_{3,3,3}$ since the cells $(1, 2)$, $(3, 1)$, and $(3, 3)$ are respectively filled with values 2, 4, and 3. We may also write this condition as $g_{1,2,2} = g_{3,1,4} = g_{3,3,3} = 1$. One solution to the instance in

Fig. 2a is depicted in Fig. 2b and corresponds to the following formula ψ:

$$\begin{aligned}\psi \equiv\ & g_{1,1,1} \wedge \neg g_{1,1,2} \wedge \neg g_{1,1,3} \wedge \neg g_{1,1,4} \\ & \wedge \neg g_{1,2,1} \wedge g_{1,2,2} \wedge \neg g_{1,2,3} \wedge \neg g_{1,2,4} \\ & \wedge \neg g_{1,3,1} \wedge \neg g_{1,3,2} \wedge \neg g_{1,3,3} \wedge g_{1,3,4} \\ & \wedge \neg g_{2,1,1} \wedge \neg g_{2,1,2} \wedge g_{2,1,3} \wedge \neg g_{2,1,4} \\ & \wedge \neg g_{2,2,1} \wedge \neg g_{2,2,2} \wedge \neg g_{2,2,3} \wedge \neg g_{2,2,4} \wedge g_{2,2,5} \\ & \wedge g_{2,3,1} \wedge \neg g_{2,3,2} \wedge \neg g_{2,3,3} \wedge \neg g_{2,3,4} \wedge \neg g_{2,3,5} \\ & \wedge \neg g_{3,1,1} \wedge \neg g_{3,1,2} \wedge \neg g_{3,1,3} \wedge g_{3,1,4} \wedge \neg g_{3,1,5} \\ & \wedge \neg g_{3,2,1} \wedge g_{3,2,2} \wedge \neg g_{3,2,3} \wedge \neg g_{3,2,4} \wedge \neg g_{3,2,5} \\ & \wedge \neg g_{3,3,1} \wedge \neg g_{3,3,2} \wedge g_{3,3,3} \wedge \neg g_{3,3,4} \wedge \neg g_{3,3,5}.\end{aligned} \tag{2}$$

The construction of ψ in (2) can be explained as follows. Notice that the instance in Fig. 2a contains two regions; the first region is the top one with $s_1 = 4$ and the second region is the bottom one with $s_2 = 5$. Each cell in the first region can be filled with any integer between 1 and 4 (inclusive), while each cell in the second region can be filled with any integer between 1 and 5 (inclusive). The filling of each cell must adhere to the rules of Suguru puzzles. In addition to the formula ψ in (2), the solution in Fig. 2b can also be described using the following interpretation for each possible atomic proposition involved:

(1) for the first region, we have $g_{1,1,1} = g_{1,2,2} = g_{1,3,4} = g_{2,1,3} = 1$ and $g_{1,1,2} = g_{1,1,3} = g_{1,1,4} = g_{1,2,1} = g_{1,2,3} = g_{1,2,4} = g_{1,3,1} = g_{1,3,2} = g_{1,3,3} = g_{2,1,1} = g_{2,1,2} = g_{2,1,4} = 0$;
(2) for the second region, we have $g_{2,2,5} = g_{2,3,1} = g_{3,1,4} = g_{3,2,2} = g_{3,3,3} = 1$ and $g_{2,2,1} = g_{2,2,2} = g_{2,2,3} = g_{2,2,4} = g_{2,3,2} = g_{2,3,3} = g_{2,3,4} = g_{2,3,5} = g_{3,1,1} = g_{3,1,2} = g_{3,1,3} = g_{3,1,5} = g_{3,2,1} = g_{3,2,3} = g_{3,2,4} = g_{3,2,5} = g_{3,3,1} = g_{3,3,2} = g_{3,3,4} = g_{3,3,5} = 0$.

For the first region in Fig. 2a, each propositional variable represents whether a cell $(i, j) \in \{(1, 1), (1, 2), (1, 3), (2, 1)\}$ can be filled with a value between 1 and 4 (inclusive). Thus, there are $4 \cdot 4 = 16$ propositional variables representing the possible cell configurations for the first region. For the second region in Fig. 2a, every propositional variable signifies whether a cell $(i, j) \in \{(2, 2), (2, 3), (3, 1), (3, 2), (3, 3)\}$ can be filled with an integer between 1 and 5 (inclusive). Hence, there are $5 \cdot 5 = 25$ propositional variables signifying the possible cell configuration for the second region. Accordingly, the total number of propositional variables needed to model the puzzle in Fig. 2 is $16 + 25 = 41$. A general analysis for the total number of propositional variables in a general Suguru puzzle will be discussed separately in Section 3.2. Notice that the formula ψ in (2) preserves the truth values of the propositional variables corresponding to the hint cells, namely $g_{1,2,2} = g_{3,1,4} = g_{3,3,3} = 1$.

To formally construct formulas relating to Suguru puzzles, we consider the following three rules, which are refined and formalized from the constraints of Suguru puzzles in Section 2.1:

(1) Every cell (i, j) in the instance contains a unique value in a region. In other words, there is some v such that (i, j) contains v and there are no two values v and v' with $v \neq v'$ such that (i, j) contains both v and v'.
(2) For every region k ($1 \leq k \leq R$) containing s_k cells, every value $v \in \{1, 2, \ldots, s_k\}$ must occur in one of the cell (i, j) within such a region.
(3) No two adjacent cells contain the same number. In other words, if (i, j) and (i', j') are two adjacent cells, then the values within (i, j) and (i', j') must be different.

Notice that the combination of the first and second rule above ensures that every cell (i, j) within a region k ($1 \leq k \leq R$) must be filled with a unique integer taken from the set $\{1, 2, \ldots, s_k\}$ where s_k denotes the number of cells in such region.

In the following sections, we discuss the encoding of each rule and analyze the size of the resulting formulas in terms of the number of clauses and literals. Measuring the size of the CNF formulas in terms of the number of clauses and literals is common in SAT-based approaches for solving a problem (see, e.g., [27], [29], [30], [31]).

3.1.1. *SAT Encoding of the First Rule: Each Cell Contains a Unique Integer in a Region.*

This section focuses on the SAT encoding of the first rule, which states that each cell (i, j) in a region k ($1 \leq k \leq R$) of size s_k must contain exactly one integer from the set $\{1, 2, \ldots, s_k\}$. We denote the propositional formula representing this rule as ϕ_1. We introduce two sub-formulas $\phi_{1,1}$ and $\phi_{1,2}$ which decompose ϕ_1 into more understandable parts. Here, $\phi_{1,1}$ ensures that each cell contains at least one integer from the set $\{1, 2, \ldots, s_k\}$ (i.e., a cell cannot be empty), whereas $\phi_{1,2}$ guarantees that no two distinct integers present within the cell (i, j) for every $1 \leq i \leq m$ and $1 \leq j \leq n$ (i.e., a cell cannot contain more than one value).

We define the formula $\phi_{1,1}$ as follows:

$$\phi_{1,1} \stackrel{\text{def}}{\equiv} \bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} \bigvee_{\substack{v=1,\\ k=r_{i,j}}}^{s_k} g_{i,j,v}. \tag{3}$$

The construction of $\phi_{1,1}$ in (3) is explained as follows. For a cell (i, j) in any region k, the clause $g_{i,j,1} \vee g_{i,j,2} \vee \cdots \vee g_{i,j,s_k}$ is introduced, which can be concisely written as $\bigvee_{v=1,k=r_{i,j}}^{s_k} g_{i,j,v}$. This clause guarantees that the cell (i, j) in region k contains at least one integer from the set $\{1, 2, \ldots, s_k\}$. Since this clause must be true for every cell (i, j) where $1 \leq i \leq m$, $1 \leq j \leq n$, and (i, j) is in region k, we obtain the formula $\phi_{1,1}$ as defined in (3).

The formula $\phi_{1,2}$ ensures that no two distinct values are present within the same cell (i, j). If this condition holds, then every cell contains at most one value.

We define the formula $\phi_{1,2}$ as follows:

$$\phi_{1,2} \stackrel{\text{def}}{\equiv} \bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} \bigwedge_{\substack{v=1,\\ k=r_{i,j}}}^{s_k-1} \bigwedge_{\substack{v'=v+1,\\ k=r_{i,j}}}^{s_k} (\neg g_{i,j,v} \vee \neg g_{i,j,v'}). \tag{4}$$

The idea for the construction of $\phi_{1,2}$ in (4) is explained as follows. We first set the rule that for a given cell (i, j), this cell cannot contain two different integers v and v' simultaneously. Notice that if (i, j) contains two different integers v and v', we have $g_{i,j,v} \wedge g_{i,j,v'}$ where $v \neq v'$. Accordingly, the formula $\neg(g_{i,j,v} \wedge g_{i,j,v'})$ with $v \neq v'$ means that there are no distinct integers v and v' satisfying both $g_{i,j,v}$ and $g_{i,j,v'}$. Since $\neg(g_{i,j,v} \wedge g_{i,j,v'}) \equiv \neg g_{i,j,v} \vee \neg g_{i,j,v'}$, we introduce the clause $\neg g_{i,j,v} \vee \neg g_{i,j,v'}$, which evaluates to false if and only if the cell (i, j) contains both integers v and v'. To include this clause for every possible combination of integer pairs (v, v') where $v \neq v'$ and $v, v' \in \{1, 2, \ldots, s_k\}$ for the cell (i, j) in region k, the following expression is used:

$$\phi'_{1,2} \stackrel{\text{def}}{\equiv} \bigwedge_{\substack{v=1,\\ k=r_{i,j}}}^{s_k-1} \bigwedge_{\substack{v'=v+1,\\ k=r_{i,j}}}^{s_k} (\neg g_{i,j,v} \vee \neg g_{i,j,v'}). \tag{5}$$

This expression is commonly known as the *pairwise* encoding [41]. Since the expression (5) must hold for every cell (i, j) in region k where $1 \leq i \leq m$ and $1 \leq j \leq n$, we obtain $\phi_{1,2}$ as defined in (4).

Finally, we arrive at ϕ_1 which combines $\phi_{1,1}$ and $\phi_{1,2}$:

$$\phi_1 \stackrel{\text{def}}{\equiv} \phi_{1,1} \wedge \phi_{1,2}. \tag{6}$$

The expression (6) asserts that each cell within a region k ($1 \leq k \leq R$) contains exactly one integer. The following lemma discusses the number of clauses and literals in ϕ_1.

Lemma 3.1. *The formula ϕ_1 in (6) consists of $mn + \sum_{k=1}^{R}(s_k^2 \cdot (s_k - 1))/2$ clauses. In addition, each clause in the sub-formula $\phi_{1,1}$ associated with a region k ($1 \leq k \leq R$) has s_k literals, while each clause in the sub-formula $\phi_{1,2}$ has two literals.*

Proof. The number of clauses in $\phi_{1,1}$ is straightforward to determine, as it is equal to the total number of cells in the puzzle, which is mn, with each cell contributing one clause. For the number of literals in each clause, notice that for each region k, we have s_k values of v from the set $\{1, 2, \ldots, s_k\}$. This means the number of literals in a clause of the form $\bigvee_{v=1,k=r_{i,j}}^{s_k} g_{i,j,v}$ is s_k.

The number of clauses in $\phi_{1,2}$ follows a different pattern. For each cell in region k where $s_k > 1$, clauses are constructed by selecting two distinct values, v and v', from the set $\{1, 2, \ldots, s_k\}$, for all possible combinations of v and v'. Consequently, each cell contributes $\binom{s_k}{2} = (s_k \cdot (s_k - 1))/2$ clauses. Considering all mn cells, the total number of clauses in $\phi_{1,2}$ can be expressed as the sum $\sum_{i=1}^{m} \sum_{j=1,k=r_{i,j}}^{n} (s_k \cdot (s_k - 1))/2$. By observing that there are s_k cells in every region k (where $1 \leq k \leq R$), we can further simplify the summation into $\sum_{k=1}^{R}(s_k^2 \cdot$

$(s_k - 1))/2$. Notice that each clause in $\phi_{1,2}$ contains two literals, i.e., $\neg g_{i,j,v}$ and $\neg g_{i,j,v'}$.

We combine the number of clauses from both $\phi_{1,1}$ and $\phi_{1,2}$ to obtain the overall number of clauses in ϕ_1, resulting in $mn + \sum_{k=1}^{R} s_k^2 \cdot (s_k - 1)/2$ clauses. Notice that this condition also holds if one of the regions contains a single cell—such a region contributes 0 clauses to the sub-formula $\phi_{1,2}$. □

3.1.2. *SAT Encoding of the Second Rule: Every Value in a Specified Range Must Be Assigned to a Cell.*

This section focuses on the SAT encoding of the second rule, which states that every value from the set $\{1, 2, \ldots, s_k\}$ (for every region k where $1 \leq k \leq R$) must be assigned to a cell. We represent this rule using the formula ϕ_2.

Efficient construction of ϕ_2 can be done as follows. The second rule emphasizes the condition that every value $v \in \{1, 2, \ldots, s_k\}$ in a region k ($1 \leq k \leq R$) containing s_k cells must occur at least once in the cell (i, j) (that belongs to the region k). Suppose we consider a region k and a value $v \in \{1, 2, \ldots, s_k\}$, if at least one cell (i, j) in such region contains v, then $g_{i,j,v}$ is true for some cell (i, j) within this region. Consequently, we have the following formula

$$\phi_2' \stackrel{\text{def}}{\equiv} \bigvee_{i=1}^{m} \bigvee_{\substack{j=1, \\ r_{i,j}=k}}^{n} g_{i,j,v}, \tag{7}$$

for a region k and for every value $v \in \{1, 2, \ldots, s_k\}$. Notice that ϕ_2' in (7) is true if and only if at least one cell in region k contains $v \in \{1, 2, \ldots, s_k\}$. To include ϕ_2' for every value v in $\{1, 2, \ldots, s_k\}$, we use the following formula

$$\begin{aligned} \phi_2'' &\stackrel{\text{def}}{\equiv} \bigwedge_{v=1}^{s_k} \phi_2' \\ &\equiv \bigwedge_{v=1}^{s_k} \bigvee_{i=1}^{m} \bigvee_{\substack{j=1, \\ r_{i,j}=k}}^{n} g_{i,j,v}. \end{aligned} \tag{8}$$

Thus, to ensure that ϕ_2'' in (8) is included for every region k where $1 \leq k \leq R$, we obtain the formula ϕ_2 as follows:

$$\begin{aligned} \phi_2 &\stackrel{\text{def}}{\equiv} \bigwedge_{k=1}^{R} \phi_2'' \\ &\equiv \bigwedge_{k=1}^{R} \bigwedge_{v=1}^{s_k} \bigvee_{i=1}^{m} \bigvee_{\substack{j=1 \\ r_{i,j}=k}}^{n} g_{i,j,v}. \end{aligned} \tag{9}$$

The following lemma discusses the number of clauses and literals in ϕ_2 in (9).

Lemma 3.2. *The formula ϕ_2 in (9) consists of $\sum_{k=1}^{R} s_k$ clauses where each clause associated with a region k has s_k literals.*

Proof. Determining the number of clauses in ϕ_2 is straightforward. In each region k, we have s_k values from the set $\{1, 2, \ldots, s_k\}$, where each value corresponds to a single clause. Hence, the total number of clauses in ϕ_2 can be obtained by summing s_k across all regions k ($1 \leq k \leq R$), which is expressed as $\sum_{k=1}^{R} s_k$.

The number of literals in each clause in ϕ_2 varies. From ϕ_2' in (7) and ϕ_2'' in (8), we know that for a clause associated with a region k ($1 \leq k \leq R$), the number of literals equals to the number of cells in such region. This is because the formula ϕ_2' represents the condition when at least one cell (i, j) in region k ($1 \leq k \leq R$) contains the value $v \in \{1, 2, \ldots, s_k\}$. Since there are s_k cells in such region, we obtain s_k as the number of literals in the aforementioned clause. □

3.1.3. *SAT Encoding of the Third Rule: No Two Adjacent Cells Contain the Same Number.*

This section delves into the SAT encoding of the third rule, which prohibits the presence of the same number in any two adjacent cells, be it horizontally, vertically, or diagonally. We denote ϕ_3 as a propositional formula representing this rule.

To break down ϕ_3 into manageable parts, we introduce eight sub-formulas, denoted by $\phi_{3,1}$, $\phi_{3,2}$, $\phi_{3,3}$, $\phi_{3,4}$, $\phi_{3,5}$, $\phi_{3,6}$, $\phi_{3,7}$, and $\phi_{3,8}$, each corresponding to one of the eight possible adjacent directions for any given cell (see Fig. 3). These sub-formulas incorporate the clause $\neg g_{i,j,v} \vee \neg g_{i',j',v}$, which states that a cell (i, j) and its adjacent cell (i', j') cannot contain the same value v. Notice that the clause $\neg g_{i,j,v} \vee \neg g_{i',j',v}$ is false when both $g_{i,j,v}$ and $g_{i',j',v}$ are true, i.e., when both cells are filled by the same value v.

1	2	3
4	(i, j)	5
6	7	8

FIGURE 3. A cell (i, j) and the position of its adjacent cells in row-major order label (i.e., the ℓ-th neighbor). Notice that some adjacent cells may not be available if they are out of the grid. Cell 1 (first neighbor) corresponds to the cell $(i - 1, j - 1)$ while cell 8 (eighth neighbor) corresponds to the cell $(i + 1, j + 1)$.

To ensure that every value $v \in \{1, 2, \ldots, s_k\}$ does not appear in two adjacent cells (i, j) and (i', j'), we consider the following formula

$$\phi_3' \stackrel{\text{def}}{\equiv} \bigwedge_{\substack{v=1, \\ k=\min\{r_{i,j}, r_{i',j'}\}}}^{s_k} (\neg g_{i,j,v} \vee \neg g_{i',j',v}). \tag{10}$$

Notice that in (10), the cell (i, j) belongs to region k, but its neighbor (i', j') does not necessarily appear in region k. The value k is set to avoid out-of-bound indices for the variable v. Although we can set $k = r_{i,j}$ and define $g_{i',j',v} = 0$ whenever the value v is larger than k, we can set $k = \min\{r_{i,j}, r_{i',j'}\}$ to minimize the number of propositional atoms in ϕ'_3. Then, to ensure that ϕ'_3 in (10) is considered for every cell (i, j) in a region k and its adjacent cell (i', j') where $1 \leq i, i' \leq m$ and $1 \leq j, j' \leq n$, we obtain the sub-formulas of the form $\phi_{3,\ell}$, where $1 \leq \ell \leq 8$, representing that the cell (i, j) in region k and its ℓ-th neighbor (according to Fig. 3) contain distinct values. The sub-formulas of the form $\phi_{3,\ell}$ are respectively defined in (11)–(18).

$$\phi_{3,1} \stackrel{\text{def}}{\equiv} \bigwedge_{i=2}^{m} \bigwedge_{j=2}^{n} \bigwedge_{\substack{v=1,\\ k=\min\{r_{i,j}, r_{i-1,j-1}\}}}^{s_k} (\neg g_{i,j,v} \vee \neg g_{i-1,j-1,v}). \tag{11}$$

$$\phi_{3,2} \stackrel{\text{def}}{\equiv} \bigwedge_{i=2}^{m} \bigwedge_{j=1}^{n} \bigwedge_{\substack{v=1,\\ k=\min\{r_{i,j}, r_{i-1,j}\}}}^{s_k} (\neg g_{i,j,v} \vee \neg g_{i-1,j,v}). \tag{12}$$

$$\phi_{3,3} \stackrel{\text{def}}{\equiv} \bigwedge_{i=2}^{m} \bigwedge_{j=1}^{n-1} \bigwedge_{\substack{v=1,\\ k=\min\{r_{i,j}, r_{i-1,j+1}\}}}^{s_k} (\neg g_{i,j,v} \vee \neg g_{i-1,j+1,v}). \tag{13}$$

$$\phi_{3,4} \stackrel{\text{def}}{\equiv} \bigwedge_{i=1}^{m} \bigwedge_{j=2}^{n} \bigwedge_{\substack{v=1,\\ k=\min\{r_{i,j}, r_{i,j-1}\}}}^{s_k} (\neg g_{i,j,v} \vee \neg g_{i,j-1,v}). \tag{14}$$

$$\phi_{3,5} \stackrel{\text{def}}{\equiv} \bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n-1} \bigwedge_{\substack{v=1,\\ k=\min\{r_{i,j}, r_{i,j+1}\}}}^{s_k} (\neg g_{i,j,v} \vee \neg g_{i,j+1,v}). \tag{15}$$

$$\phi_{3,6} \stackrel{\text{def}}{\equiv} \bigwedge_{i=1}^{m-1} \bigwedge_{j=2}^{n} \bigwedge_{\substack{v=1,\\ k=\min\{r_{i,j}, r_{i+1,j-1}\}}}^{s_k} (\neg g_{i,j,v} \vee \neg g_{i+1,j-1,v}). \tag{16}$$

$$\phi_{3,7} \stackrel{\text{def}}{\equiv} \bigwedge_{i=1}^{m-1} \bigwedge_{j=1}^{n} \bigwedge_{\substack{v=1,\\ k=\min\{r_{i,j}, r_{i+1,j}\}}}^{s_k} (\neg g_{i,j,v} \vee \neg g_{i+1,j,v}). \tag{17}$$

$$\phi_{3,8} \stackrel{\text{def}}{\equiv} \bigwedge_{i=1}^{m-1} \bigwedge_{j=1}^{n-1} \bigwedge_{\substack{v=1,\\ k=\min\{r_{i,j}, r_{i+1,j+1}\}}}^{s_k} (\neg g_{i,j,v} \vee \neg g_{i+1,j+1,v}). \tag{18}$$

Notice that the sub-formulas $\phi_{3,\ell}$ in (11)–(18) exhaustively represent that the values in two adjacent cells (i, j) and (i', j') must be different. For example, the sub-formula $\phi_{3,1}$ means that the cell (i, j) and $(i-1, j-1)$ are filled with different values. The upper and lower bounds for the indices i and j in (11)–(18) are adjusted to avoid the cells that are out of the grid (e.g., $i < 1$, $j < 1$, $i > m$, or $j > n$).

Finally, we obtain the formula ϕ_3 as defined in (19), which conjuncts all sub-formulas in (11)–(18):

$$\phi_3 \stackrel{\text{def}}{\equiv} \phi_{3,1} \wedge \phi_{3,2} \wedge \phi_{3,3} \wedge \phi_{3,4} \wedge \phi_{3,5} \wedge \phi_{3,6} \wedge \phi_{3,7} \wedge \phi_{3,8}. \tag{19}$$

This formula asserts that no two adjacent cells contain identical values. For practical purpose, we may define $g_{i,j,v} = 0$ for any v if $i < 1$, $j < 1$, $i > m$, or $j > n$. However, the implementation detail of the SAT solver might allow us to exclude the clause that corresponds to the cells that are not within the grid.

Determining the exact number of clauses in ϕ_3 is a little bit cumbersome because the number of clauses corresponding to a cell (i, j) depends on the number of neighbors of (i, j). A cell might have three, five, or eight neighbor cells. Nevertheless, we can estimate an upper bound for the number of clauses by considering the fact that an empty cell has at most eight neighbors. The following lemma discusses the upper bound for the number of clauses in ϕ_3 and the number of literals in each clause.

Lemma 3.3. *The formula ϕ_3 in* (19) *consists of no more than $\sum_{k=1}^{R} 8 \cdot s_k^2$ clauses where each clause has two literals.*

Proof. We first analyze the number of literals for each clause in ϕ_3 defined in (19). Notice that ϕ_3 is a conjunction of sub-formulas of the form $\phi_{3,\ell}$ defined in (11)–(18). Each of the sub-formulas $\phi_{3,\ell}$ is instantiated from the formula ϕ'_3 in (10) containing two literals. As a result, each clause in ϕ_3 has two literals.

For the number of clauses, notice that a cell has at most eight neighbors (see Fig. 3). From the definition of ϕ'_3 in (10), for each pair of adjacent cells (i, j) and (i', j'), the number of clauses is at most given by s_k, where $k = \min\{r_{i,j}, r_{i',j'}\} \leq r_{i,j}$. Subsequently, by considering all possible pairs of adjacent cells (i, j) and (i', j'), ϕ_3 contains no more than $\sum_{i=1}^{m} \sum_{j=1,k=r_{i,j}}^{n} 8 \cdot s_k$ clauses. By observing that there are s_k cells in every region k (where $1 \leq k \leq R$), we can further simplify this expression to $\sum_{k=1}^{R} 8 \cdot s_k^2$. □

3.2. **The Number of Variables and Clauses**

This section examines the number of variables and clauses required to solve a Suguru puzzle of size $m \times n$ using a SAT-based approach considering the SAT encodings mentioned earlier. The significance of this examination lies in its connection to the asymptotic time complexity of a SAT-based approach, which is determined by the number of variables and clauses present in the resulting propositional formulas [42]. Our analysis is also motivated by recent works (e.g., [30], [31]), which consider the number of variables and clauses to evaluate their theoretical efficiency. The following theorem discusses this examination.

Theorem 3.4. *The total number of clauses in formulas* (6), (9), *and* (19) *required for solving an $m \times n$ Suguru puzzle containing R regions, where the size of each region k ($1 \leq k \leq R$) is s_k, is bounded above by $mn + \sum_{k=1}^{R} \left[(s_k^2 \cdot (s_k - 1))/2 + 8 \cdot s_k^2 + s_k\right]$.*

Moreover, the number of variables for solving the aforementioned Suguru puzzle is $\sum_{k=1}^{R} s_k^2$.

Proof. We determine the number of clauses and variables by analyzing ϕ_1, ϕ_2, and ϕ_3. Notice that each of these formulas utilizes all variables $g_{i,j,v}$ for $1 \leq i \leq m, 1 \leq j \leq n$, and $1 \leq v \leq s_k$ where $k = r_{i,j}$.

First, we calculate the number of variables by considering the number of possible values for each cell (i, j) in the puzzle. For a region k $(1 \leq k \leq R)$ consisting of s_k cells, there are s_k possible values that can be filled into each cell. Therefore, each region k requires s_k^2 variables to represent all possible value assignments; each assignment corresponds to a specific configuration. To obtain the total number of variables for all R regions, we sum up the variables needed for each region, resulting in $\sum_{k=1}^{R} s_k^2$.

As for the number of clauses, we use the result from the previous lemmas for each constraint encoding:

(1) from Theorem 3.1, ϕ_1 consists of $mn + \sum_{k=1}^{R}(s_k^2 \cdot (s_k - 1))/2$ clauses;
(2) from Theorem 3.2, ϕ_2 consists of $\sum_{k=1}^{R} s_k$ clauses;
(3) from Theorem 3.3, ϕ_3 consists of at most $\sum_{k=1}^{R} 8 \cdot s_k^2$ clauses.

We obtain the upper bound for the total number of clauses in the resulting formula by adding the number of clauses from each lemma. □

The following corollary follows from Theorem 3.4.

Corollary 3.5. *Suppose we consider an $m \times n$ Suguru puzzle instance with R regions where each region contains s_k cells $(1 \leq k \leq R)$. The total number of variables in the resulting formula for representing such an instance is bounded above by $O(m^2n^2)$, while the total number of clauses in the resulting formula for representing the same instance is bounded above by $O(m^3n^3)$.*

Proof. For an $m \times n$ Suguru puzzle instance with R regions where each region contains s_k cells $(1 \leq k \leq R)$, we have $R \leq mn$ and $s_k \leq mn$ for each $1 \leq k \leq R$. We also notice that $\sum_{k=1}^{R} s_k = mn$ since s_k represents the number of cells in a region k $(1 \leq k \leq R)$ while mn represents the total number of cells in the grid. Since $s_k \geq 1$, we also have

$$\sum_{k=1}^{R} s_k^2 \leq \left(\sum_{k=1}^{R} s_k\right)^2 = (mn)^2 = m^2n^2, \text{ and} \tag{20}$$

$$\sum_{k=1}^{R} s_k^3 \leq \left(\sum_{k=1}^{R} s_k\right)^3 = (mn)^3 = m^3n^3. \tag{21}$$

By Theorem 3.4, the total number of variables for solving the instance is $\sum_{k=1}^{R} s_k^2$, which is bounded above by $O(m^2n^2)$ according to (20). To obtain the upper bound for the number of clauses, we notice that $s_k^2(s_k - 1) = s_k^3 - s_k^2 < s_k^3$ if

$s_k \geq 1$ and observe the following inequality

$$\begin{aligned} mn + \sum_{k=1}^{R} \left[(s_k^2 \cdot (s_k - 1))/2 + 8 \cdot s_k^2 + s_k\right] &= mn + \sum_{k=1}^{R} \left[(s_k^3 - s_k^2)/2 + 8 \cdot s_k^2 + s_k\right] \\ &< mn + \sum_{k=1}^{R} \left[s_k^3 + 8 \cdot s_k^3 + s_k^3\right] \\ &= mn + 10 \sum_{k=1}^{R} s_k^3 \\ &< mn + 10 \cdot m^3 n^3 \text{ (by (21))}, \end{aligned}$$

that is, the number of clauses in all formulas is bounded above by $O(m^3n^3)$. □

The results in Theorem 3.5 infer that the total number of variables and clauses in our proposed encodings are *polynomially proportional* to the puzzle's size. In particular, when considering a Suguru puzzle of size $m \times n$, the total number of variables is bounded above by $O(m^2n^2)$, while the total number of clauses is bounded above by $O(m^3n^3)$. These imply that it is possible to transform a Suguru puzzle instance to a SAT problem instance using a polynomial-size transformation with respect to the input size. These theoretical results suggest that the SAT solver can be used to solve the Suguru puzzle efficiently if the construction of variables and clauses is made appropriately.

It is also important to note that the number of hints does not asymptotically affect the number of variables and clauses in the resulting formula. As stated earlier at the beginning of Section 3.1, hints are represented as a conjunction of several atomic propositions associated with the pre-filled cells. Despite not affecting the asymptotic upper bound for the number of variables and clauses, the number of hints might potentially reduce the search space for the SAT solver in our SAT-based approach.

4. Computational Experiments: Design, Results, and Analysis

The results in Theorem 3.5 suggest that an instance of Suguru puzzles can be transformed into a SAT problem instance efficiently, thereby allowing one to find the solution to a Suguru instance using the SAT solver effectively. The main purpose of this section is to describe the design, results, and analysis of our computational experiments. Experiments were conducted to evaluate the running times of our proposed SAT-based approach against various test cases. These running times were also contrasted with the running times of the previous backtracking algorithm proposed in [15] to provide a performance benchmark and for comparative purposes.

The experiments used the same computational environment as in [15]. Namely, we used a standard laptop with a 64-bit Windows 10 operating system equipped with an Intel(R) Core(TM) i7-1070H processor at 2.60 GHz and 16 GB of RAM. The SAT-based approach was implemented using MiniSat—a lightweight C++ library that provides a SAT solver interface and has historically been one of the most

influential and widely studied SAT solvers. Although MiniSat is no longer actively maintained, it remains a popular choice for research and education purposes due to its simplicity and stable API. We selected MiniSat because it is implemented in C++, the same language used in [15] for the backtracking approach, ensuring objective and consistent performance evaluation between the two methods. MiniSat also provides a stable baseline with numerous examples and tools built around it. One should note that our main focus is on the SAT encoding rather than solver performance. However, we acknowledge that modern SAT solvers such as Glucose [43], CaDiCaL [44], or Kissat [45] could potentially offer improved efficiency. Suguru rules and hints were represented using propositional formulas that were translated into MiniSat inputs.

As the experiments in [15], we considered the comprehensive 180 test cases collected from [46]. These test cases include Suguru puzzles of size $n \times n$ where $6 \leq n \leq 10$. Of these test cases, 52 of them are of size 8×8. In addition to these test cases, we also randomly generated 50 test cases for further evaluation. These additional test cases are of size $n \times n$ where $11 \leq n \leq 15$, hence introducing puzzles of larger sizes to the experiments. The hint cells in the additional test cases constitute no more than 40% of the total number of cells. All instances in the test cases, both from [46] and the additional ones, are guaranteed to have a unique solution.

The procedure for generating additional test cases with guaranteed unique solutions is as follows. We first generate random Suguru board structures of size $n \times n$ where $11 \leq n \leq 15$ using a randomized breadth-first search algorithm to create connected regions of $4 - 8$ cells in each structure. For each blank grid, we employ our SAT-based solver to find a complete solution, then apply a greedy hint minimization algorithm that iteratively removes filled cells while preserving solution uniqueness.

Uniqueness verification for an additional test case is performed using exhaustive SAT-based checking. The algorithm removes the hints one by one. After each hint removal, we solve the SAT instance to obtain the first solution, then add a constraint to exclude this solution, and attempt to solve the puzzle again. To check for another solution, we convert the first solution into a propositional formula, negate it, use the negated formula as a constraint, and try to find the second solution. If a second solution exists, the removed hint is restored as essential. The algorithm continues to remove hints until removing any additional one would result in a puzzle with multiple solutions. This way, the final puzzle is guaranteed to have exactly one solution. Our algorithm is also set so that the final puzzle contains at most 40% of cells as hints.

From multiple generation attempts across different board sizes, we successfully created 50 instances with a mathematically guaranteed unique solution. For each $n \times n$ board where $11 \leq n \leq 15$, we were able to construct 10 Suguru instances that have a unique solution with at most 40% of their cells as hints. The generation process involves multiple SAT solver calls for uniqueness verification, requiring substantial computational time that is separate from the puzzle-solving

time, which is the main concern of our experimental investigation. The technical details of the method to generate a Suguru puzzle with a unique solution are described in `https://github.com/abcqwq/suguru-sat/tree/main/additional_testcases_generator`.

We performed ten independent runs for each test case to obtain objective and accurate results regarding the running times for solving instances of Suguru puzzles. The measurements represent real elapsed times (wall clock times) observed during execution. It should be noted that the reported solving times measure only the time needed to solve the final puzzle with hints and do not include the time required to generate the puzzle. The running time data were then classified by grid size, and the average running time for each classification of Suguru puzzle instances was calculated. The C++ scripts, test cases, raw experimental results, and other relevant documents related to the experiment are available at `https://github.com/abcqwq/suguru-sat`. Our interactive SAT-based Suguru puzzle solver is available at `https://github.com/abcqwq/interactive-suguru-playground`.

To provide an objective and comprehensive analysis, we also re-run the experiment in solving the Suguru puzzles using the backtracking approach explained in [15]. Table 1 summarizes and contrasts the running times needed for solving the Suguru puzzles using the backtracking approach in [15] and our proposed SAT-based approach. Although there are some differences between the results in [15, Table 1] and those in Table 1 for the running times of the backtracking algorithm, their differences are less than 10% and can be considered negligible. These differences might occur due to some background processes in the computational device during the experiment. To provide more intuitive insight into the experimental results, we present and contrast the distribution of the running times for both backtracking and SAT-based approaches using boxplots in Fig. 4.

From Table 1, we notice that the SAT-based approach solves all instances in less than 12 milliseconds. The data in Table 1 and its representation in Fig. 4 indicate that the average running times of the SAT-based approach are less than that of the backtracking algorithm for larger puzzles, with crossover point as low as $n = 11$. The running time variances of the SAT-based approach are also significantly smaller than those of the backtracking algorithm as indicated in Fig. 4. As discussed in Section 3.2, the running time of our SAT-based approach does not directly depend on the number of hints—the main contributor to this running time is the size of the puzzle. On the other hand, the running time of the backtracking algorithm in [15] depends on the number of regions and hints, specifically the asymptotic running time for solving an $m \times n$ Suguru instance containing R regions and H hints is $O(R \cdot (mn - H + 2)!)$ (cf. [15, Corollary 1]).

Although the average running times for the SAT-based approach for solving the $n \times n$ puzzles where $6 \leq n \leq 10$ are higher than those of the backtracking algorithm (with the exception for the 8×8 puzzles), the maximum running times for solving the instances for SAT-based approach are less than those of the backtracking algorithm for the puzzles of sizes 6×6, 8×8, and 9×9. Moreover, one challenging instance of size 8×8 that takes almost 480 milliseconds to be solved

TABLE 1. Running times (in milliseconds) for 230 test cases taken by the backtracking algorithm (denoted by BT) proposed in [15] and SAT-based approach (denoted by SAT) proposed in this paper. The min., max., and avg. respectively denote the minimum, maximum, and average running times for solving the puzzles for particular instances. The number of instances for each dimension is denoted by # TC.

Puzzle Size	**# TC**	**Min.**		**Max.**		**Avg.**	
		BT	**SAT**	**BT**	**SAT**	**BT**	**SAT**
6×6	42	0.024	0.837	3.677	1.053	0.243	0.918
7×7	24	0.032	1.275	0.595	1.574	0.119	1.402
8×8	52	0.038	1.714	474.484	2.136	11.761	1.903
9×9	24	0.093	2.596	21.586	3.038	1.837	2.819
10×10	38	0.137	3.501	3.876	4.020	1.084	3.703
11×11	10	1.294	4.514	517.712	5.536	111.622	5.008
12×12	10	3.478	5.652	684.979	6.776	193.156	6.007
13×13	10	7.768	7.113	19764.900	7.877	4146.783	7.336
14×14	10	24.439	8.078	88028.100	9.163	18835.462	8.606
15×15	10	61.329	9.119	573389.000	11.067	94464.486	10.000

by the backtracking algorithm can be solved in under 2.5 milliseconds by the SAT-based approach. This challenging instance—which is also discussed in [15, Fig. 7]—is depicted in Fig. 5. In terms of the minimum running time, the data in Table 1 shows that the SAT-based approach outperforms the backtracking algorithm for solving Suguru instances of sizes $n \times n$ where $n \geq 13$.

Based on the average running times data in Table 1—we see that, on average—our proposed SAT-based approach is at least around 20 times faster than the backtracking algorithm in solving the $n \times n$ puzzles for $11 \leq n \leq 15$. Furthermore, based on the minimum running times data in Table 1, we see that the SAT-based approach is at least three times faster than the backtracking algorithm in solving the $n \times n$ puzzles for $n \in \{14, 15\}$. Interesting results occur for the 15×15 puzzles, in which the SAT-based approach is at least six times faster than the backtracking algorithm for solving all instances and is up to 50 000 times faster than the backtracking algorithm for solving some specific instance.

One should note that unlike the backtracking technique proposed in [15], the number of regions, the number of hints cells, and the hint cells' arrangement do not directly affect the asymptotic running time of our SAT-based approach. Based on these experimental results, we might infer that our SAT-based approach is more

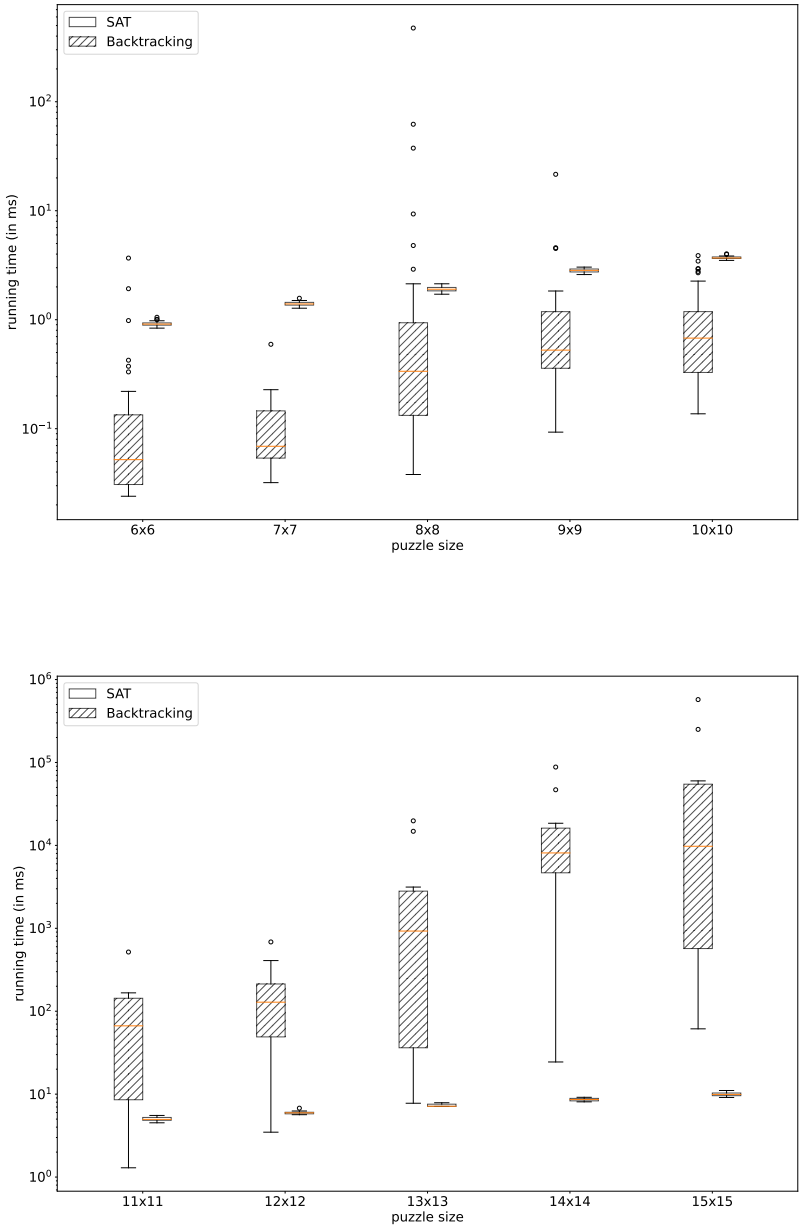
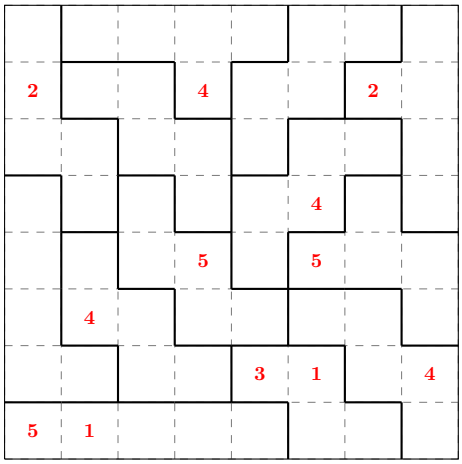


FIGURE 4. Boxplots illustrating the running time distributions of the backtracking algorithm and the SAT-based approach. For each point on the horizontal axis, the left-side dashed boxplot represents the running time distributions for the backtracking algorithm, while the right-side white boxplot represents the running time distributions for the SAT-based approach.

efficient for solving larger Suguru puzzles—although it is not always efficient in solving the smaller ones.



(A) A challenging instance of a Suguru puzzle.

4	1	2	5	3	1	5	1
2	5	3	**4**	2	4	**2**	3
3	1	2	1	3	5	1	4
4	5	3	4	2	**4**	3	5
1	2	1	**5**	3	**5**	1	4
5	**4**	3	2	4	2	3	2
3	2	5	1	**3**	**1**	5	**4**
5	**1**	3	4	2	4	2	1

(B) The solution to the Suguru instance in Fig. 5a.

FIGURE 5. A challenging Suguru instance in [15, Fig. 7], bold numbers represent the hints. The SAT-based approach can solve the instance in Fig. 5a in less than 2.5 milliseconds (i.e., 1.849 ms). Such an instance also provides an example in which the SAT-based approach outperforms the backtracking technique by a factor of approximately 255 in terms of running time.

5. Concluding Remarks and Open Problems

In this paper, we have demonstrated an efficient SAT-based technique for solving general Suguru puzzles. We express the rules of Suguru puzzles in propositional formulas in CNF and discuss their properties regarding the number of variables, clauses, and literals. In Theorem 3.5, we show that the number of variables and clauses in the resulting formulas is bounded above by some polynomials representing the puzzles' sizes. Thus, our proposed encoding is polynomially proportional to the dimensions of the puzzles, which allows for the efficient transformation of Suguru instances into SAT problem instances. This encoding also provides a technique for reducing Suguru puzzles into SAT problems.

We utilize MiniSat, implemented in C++, for our experiments. Although MiniSat is no longer actively maintained, it remains widely used in research and educational contexts. For future research, we recommend implementing our SAT-based approach using more efficient SAT solvers such as Glucose [43], CaDiCaL [44], or Kissat [45]. The results from such experiments could serve as benchmarks for evaluating algorithmic strategies in solving the Suguru puzzle.

The summary of experimental results in Table 1 and Fig. 4 suggests that our proposed SAT-based approach outperforms the backtracking technique for solving larger puzzles in terms of running time, although it is not always efficient for solving smaller puzzles. We obtained a crossover point as low as $n = 11$ for the average running time—meaning that, on average, solving $n \times n$ Suguru puzzles using the SAT-based approach is faster than the backtracking technique for $n \geq 11$. In terms of the maximum running time, we also see that the SAT-based approach is always more efficient than the backtracking technique for solving $n \times n$ puzzles for $n \geq 11$. Moreover, based on the minimum running time, we see that the SAT-based approach is always faster than the backtracking one for solving $n \times n$ puzzles for $n \geq 13$. According to the experimental results, the running time variances of our proposed SAT-based technique are much smaller than those of the backtracking technique. These empirical outcomes support the theoretical result discussed in Section 3, which states that the main contributor to the running time of the SAT-based approach is the dimension of the puzzle. Unlike the backtracking approach discussed in [15], the running time of our SAT-based approach does not directly depend on the number of hints, the positions of the hints, and the arrangements of the cells in a region. One should note that, despite the efficiency exhibited by the SAT-based approach for solving Suguru instances of larger sizes, the backtracking technique still prevails in solving smaller puzzles.

There are some interesting open problems related to solving Suguru puzzles. First, we do not prove the optimality of our SAT encoding technique. It might be possible to construct a SAT encoding that is more efficient (in terms of the number of variables and clauses) than our proposed encoding in Section 3.1. A possible improvement can be made by considering a more efficient SAT encoding for the third rule, stating that no adjacent cells contain the same integers. Second, our experimental results are limited to $n \times n$ Suguru puzzles, even though the puzzles can be defined over rectangular grids $m \times n$ where $m \neq n$. The main reason for this is because our test cases are mainly taken from [46] (which is also used in [15]). One can consider additional experiments that use more diverse test cases by employing Suguru instances of size $m \times n$ where $m \neq n$, and then can investigate the resulting empirical consequences concerning the theoretical results in Theorem 3.5. Third, our SAT-based algorithm can be used to construct a Suguru puzzle generation algorithm described in Section 4. Our current puzzle generation algorithm employs an exhaustive SAT-based checking approach, which requires a substantial amount of time to generate a puzzle with a unique solution. To our knowledge, investigations into the technique for constructing a Suguru puzzle with a unique solution are still underexplored. Finally, our SAT-based approach can also be modified to solve Suguru puzzle-related problems, such as counting the number of solutions for a given Suguru instance. To our knowledge, theoretical and computational problems related to counting the number of solutions for a Suguru puzzle instance remain an open problem.

Acknowledgment. The authors would like to thank the anonymous reviewer(s) who provided the valuable feedback. Both authors are supported by Telkom University at the beginning of the research.

References

[1] N. Inaba, *Number block*, http://inabapuzzle.com/honkaku/nblock.html, Accessed: 2023-06-19, Jul. 2001.

[2] L. Robert, D. Miyahara, P. Lafourcade, L. Libralesso, and T. Mizuki, "Physical zero-knowledge proof and NP-completeness proof of Suguru puzzle," *Information and Computation*, vol. 285, p. 104 858, 2022. DOI: https://doi.org/10.1016/j.ic.2021.104858.

[3] C. Iwamoto and T. Ide, "Calculation Solitaire is NP-Complete," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 106, no. 3, pp. 328–332, 2023. DOI: https://doi.org/10.1587/transinf.2022fcl0002.

[4] C. Iwamoto and T. Tokunaga, "Choco Banana is NP-complete," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 107, no. 9, pp. 1488–1491, 2024. DOI: https://doi.org/10.1587/transfun.2023dml0001.

[5] C. Iwamoto and T. Ibusuki, "Polynomial-time reductions from 3SAT to Kurotto and Juosan puzzles," *IEICE Transactions on Information and Systems*, vol. 103, no. 3, pp. 500–505, 2020. DOI: https://doi.org/10.1587/transinf.2019fcp0004.

[6] J. Bosboom, E. D. Demaine, M. L. Demaine, A. Hesterberg, R. Kimball, and J. Kopinsky, "Path puzzles: Discrete tomography with a path constraint is hard," *Graphs and Combinatorics*, vol. 36, no. 2, pp. 251–267, 2020. DOI: https://doi.org/10.1007/s00373-019-02092-5.

[7] A. Adler, J. Bosboom, E. D. Demaine, M. L. Demaine, Q. C. Liu, and J. Lynch, "Tatamibari is NP-Complete," in *10th International Conference on Fun with Algorithms (FUN 2021)*, M. Farach-Colton, G. Prencipe, and R. Uehara, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 157, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 1:1–1:24, ISBN: 978-3-95977-145-0. DOI: https://doi.org/10.4230/LIPIcs.FUN.2021.1.

[8] C. Iwamoto and T. Ide, "Five Cells and Tilepaint are NP-Complete," *IEICE Transcations on Information and Systems*, vol. 105, no. 3, pp. 508–516, 2022. DOI: https://doi.org/10.1587/transinf.2021fcp0001.

[9] E. D. Demaine, J. Lynch, M. Rudoy, and Y. Uno, "Yin-Yang Puzzles are NP-complete," in *33rd Canadian Conference on Computational Geometry (CCCG) 2021*, 2021.

[10] S. Saha and E. D. Demaine, "ZHED is NP-complete," in *Proceedings of the 34th Canadian Conference on Computational Geometry (CCCG 2022)*, 2022.

[11] E. C. Reinhard, M. Arzaki, and G. S. Wulandari, "Solving Tatamibari Puzzle Using Exhaustive Search Approach," *Indonesia Journal on Computing (Indo-JC)*, vol. 7, no. 3, pp. 53–80, 2022.

[12] M. I. Putra, M. Arzaki, and G. S. Wulandari, "Solving Yin-Yang Puzzles Using Exhaustive Search and Prune-and-Search Algorithms," *(IJCSAM) International Journal of Computing Science and Applied Mathematics*, vol. 8, no. 2, pp. 52–65, 2022. DOI: https://doi.org/10.12962/j24775401.v8i2.13720.

[13] M. T. Ammar, M. Arzaki, and G. S. Wulandari, "Note on Algorithmic Investigations of Juosan Puzzles," *Jurnal Ilmu Komputer dan Informasi*, vol. 17, no. 1, pp. 19–35, 2024. DOI: https://doi.org/10.21609/jiki.v17i1.1184.

[14] J. E. Sakti, M. Arzaki, and G. S. Wulandari, "A Backtracking Approach for Solving Path Puzzles," *Journal of Fundamental Mathematics and Applications (JFMA)*, vol. 6, no. 2, pp. 117–135, 2023. DOI: https://doi.org/10.14710/jfma.v6i2.18155.

[15] B. Butrahandisya, M. Arzaki, and G. S. Wulandari, "Elementary Algorithmic Methods for Solving Suguru Puzzles," *(IJCSAM) International Journal of Computing Science and Applied Mathematics*, vol. 10, no. 1, pp. 12–26, 2024. DOI: https://doi.org/10.12962/j24775401.v10i1.17249.

[16] V. A. Fridolin, M. Arzaki, and G. S. Wulandari, "Elementary Search-based Algorithms for Solving Tilepaint Puzzles," *Indonesia Journal on Computing (Indo-JC)*, vol. 8, no. 2, pp. 36–64, 2023.

[17] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, 1971, pp. 151–158. DOI: https://doi.org/10.7551/mitpress/12274.003.0036.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 4th Ed. MIT press, 2022.

[19] R. M. Karp, "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, Eds. Boston, MA: Springer US, 1972, pp. 85–103, ISBN: 978-1-4684-2001-2. DOI: https://doi.org/10.1007/978-1-4684-2001-2_9.

[20] M. Sipser, *Introduction to the Theory of Computation, 3rd Edition*. Cengage Learning, 2013.

[21] H. A. Kautz, B. Selman, et al., "Planning as Satisfiability," in *ECAI*, Citeseer, vol. 92, 1992, pp. 359–363.

[22] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe, "Theorem proving using lazy proof explication," in *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15*, Springer, 2003, pp. 355–367. DOI: https://doi.org/10.1007/978-3-540-45069-6_34.

[23] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.

[24] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002. DOI: https://doi.org/10.1109/tcad.2002.804386.

[25] Z. Lei and S. Cai, "Solving (Weighted) Partial MaxSAT by Dynamic Local Search for SAT," in *IJCAI*, vol. 7, 2018, pp. 1346–52. DOI: https://doi.org/10.24963/ijcai.2018/187.

[26] T. Balyo, P. Sanders, and C. Sinz, "HordeSat: A massively parallel portfolio SAT solver," in *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*, Springer, 2015, pp. 156–172. DOI: https://doi.org/10.1007/978-3-319-24318-4_12.

[27] P. Utomo and R. Pellikaan, "Binary puzzle as a SAT problem," in *Proceedings of the 2017 Symposium on Information Theory and Signal Processing, Benelux*, 2017, pp. 223–229.

[28] C. Ansótegui, R. Béjar, C. Fernandez, and C. Mateu, "Edge matching puzzles as hard SAT/CSP benchmarks (extended version)," Technical Report TR-1-08, Dept. of Computer Science, Universitat de Lleida, Tech. Rep., 2008.

[29] A. M. Myat, K. K. Htwe, and N. Funabiki, "Fill-a-pix puzzle as a SAT problem," in *2019 International Conference on Advanced Information Technologies (ICAIT)*, IEEE, 2019, pp. 244–249. DOI: https://doi.org/10.1109/aitc.2019.8920898.

[30] M. T. Ammar, M. Arzaki, and G. S. Wulandari, "Efficient SAT-Based Approach for Solving Juosan Puzzles," in *International Conference on Mathematics: Pure, Applied and Computation*, Springer, 2023, pp. 211–226. DOI: https://doi.org/10.1007/978-981-97-2136-8_16.

[31] J. E. Sakti, M. Arzaki, and G. S. Wulandari, "Modeling Path Puzzles as SAT Problems and How to Solve Them," in *International Conference on Mathematics: Pure, Applied and Computation*, Springer, 2023, pp. 227–245. DOI: https://doi.org/10.1007/978-981-97-2136-8_17.

[32] L. Kolijn, "Generating and Solving Skyscrapers Puzzles Using a SAT Solver," Bachelor Thesis, Radboud University, 2022.

[33] I. Lynce and J. Ouaknine, "Sudoku as a SAT Problem," in *AI&M*, 2006.

[34] U. Pfeiffer, T. Karnagel, and G. Scheffler, "A Sudoku-Solver for Large Puzzles using SAT," in *LPAR short papers (Yogyakarta)*, 2010, pp. 52–57. DOI: https://doi.org/10.29007/79mc.

[35] T. Weber, "A SAT-based Sudoku solver," in *LPAR*, 2005, pp. 11–15.

[36] C. Bright, J. Gerhard, I. Kotsireas, and V. Ganesh, "Effective problem solving using SAT solvers," in *Maple Conference*, Springer, 2019, pp. 205–219. DOI: https://doi.org/10.1007/978-3-030-41258-6_15.

[37] M. Ben-Ari, *Mathematical Logic for Computer Science, 3rd Edition*. Springer Science & Business Media, 2012.

[38] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems, 2nd Edition*. Cambridge university press, 2004.

[39] T. D. Hansen, H. Kaplan, O. Zamir, and U. Zwick, "Faster k-sat algorithms using biased-PPSZ," in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, 2019, pp. 578–589. DOI: https://doi.org/10.1145/3313276.3316359.

[40] K. H. Rosen, *Discrete Mathematics and Its Applications*, 8th Ed. McGraw-Hill Higher Education, 2019, ISBN: 0072424346.

[41] S. Hölldobler and V.-H. Nguyen, "An efficient encoding of the at-most-one constraint," *Technical Report 1304. Technische Universitäat Dresden*, 2013.

[42] M. Hořeňovský, *Modern SAT solvers: fast, neat and underused (part 3 of N)*, https://codingnest.com/modern-sat-solvers-fast-neat-and-underused-part-3-of-n/, Accessed: 2023-5-22, Apr. 2019.

[43] G. Audemard and L. Simon, "Predicting Learnt Clauses Quality in Modern SAT Solvers," in *IJCAI*, vol. 9, 2009, pp. 399–404.

[44] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "CaDiCaL 2.0," in *International Conference on Computer Aided Verification*, Springer, 2024, pp. 133–152.

[45] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., ser. Department of Computer Science Report Series B, vol. B-2020-1, University of Helsinki, 2020, pp. 51–53.

[46] O. Janko, *Suguru*, https://www.janko.at/Raetsel/Suguru/index.htm, Accessed: 2022-10-11, Oct. 2022.