Elementary Algorithmic Methods for Solving Suguru Puzzles

Butrahandisya, Muhammad Arzaki, and Gia Septiana Wulandari

***Abstract*—We discuss elementary algorithmic aspects of the Suguru puzzle—a single-player paper-and-pencil puzzle introduced in 2001 and confirmed NP-complete by Robert et al. in 2022 [1]. We propose a backtracking algorithm with pruning optimizations for solving an $m \times n$ Suguru puzzles containing R regions and H hint cells in $O(R \cdot (mn - H + 2)!)$ time. Despite this factorial asymptotic upper bound, a C++ implementation of our proposed algorithm successfully solved all Suguru instances with no more than 100 cells using a personal computer in less than 0.5 seconds. We also prove that any Suguru instance of size $m \times n$ with either $m = 1$ or $n = 1$ can be solved in linear time regarding of the puzzle size. Finally, we provide an upper bound for the number of solutions to such tractable instances.**

***Index Terms*—asymptotic analysis, backtracking, Suguru puzzle, tractable subproblems**

I. Introduction

Suguru (also known as Nanbaburokku) is a single-player pencil-and-paper puzzle invented by Naoki Inaba, a prominent Japanese puzzle designer. It first appeared in 2001 [2] and was recently proven NP-complete by Robert et al. in 2022 [1]. Like the famous Sudoku, the player must fill the empty cells in a rectangular grid, satisfying some puzzle rules. The game is played on an $m \times n$ grid partitioned into regions. A region is a collection of orthogonally connected cells. The goal is to fill all cells with numbers such that:

1) no two cells in a region can contain the same number;
2) no two adjacent cells, either orthogonally or diagonally, can contain the same number;
3) a number in a cell must be between 1 and the size of the region it belongs to, where the size of a region is defined as the number of cells in it.

Puzzles have long been regarded as captivating mental challenges that have entertained and engaged individuals throughout history. They provide leisure and diversion opportunities and stimulate cognitive skills such as critical thinking and problem-solving [3]. Moreover, theoretical aspects of puzzles have garnered substantial interest from the scientific community in the last twenty years owing to their profound links with crucial problems in mathematics and the theory of computation, resulting in extensive investigations into their mathematical and computational aspects (see [4]–[6] for extensive investigations). Furthermore, a variety of paper-and-pencil-based games have been confirmed NP-complete, including but not limited to (in chronological order): Nonogram (1996) [7], Sudoku (2003) [8], Nurikabe (2004) [9], Heyawake (2007) [10], Hashiwokakero (2009) [11], Kurodoko (2012) [12], Shikaku and Ripple Effect (2013) [13], Yosenabe (2014) [14], Fillmat (2015) [15], Dosun-Fuwari (2018) [16], Tatamibari (2020) [17], Kurotto and Juosan (2020) [18], Yin-Yang (2021) [19], Tilepaint (2022) [20], and Suguru (2022) [1].

The NP-completeness of Suguru puzzles implies the existence of a polynomial-time verification procedure for checking whether an arbitrary configuration is a solution to a Suguru instance. However, solving a Suguru puzzle remains an exponential-time task because no known polynomial-time algorithm exists for any NP-complete problem. Moreover, formal algorithmic investigation for solving Suguru puzzles has been relatively limited as it has only recently proven NP-complete. Investigations on elementary algorithmic methods such as the exhaustive search and prune-and-search—which utilizes a similar approach to the methods used in this paper—have been carried out on puzzles such as Yin-Yang [21], Tatamibari [22], Tilepaint [23], Path Puzzles [24], and Juosan Puzzles [25]. More advanced techniques are also available for solving NP-complete puzzles, such as SAT solvers [26], [27] and the deep learning method [28].

This paper discusses an elementary approach, the backtracking method, enhanced with pruning optimizations. We demonstrate that this approach can solve any Suguru puzzle, with the caveat that the solving time increases in factorial factor in terms of the puzzle size and the number of hints. In addition to this, we delve into the exploration of a tractable variant of the Suguru puzzle. Investigating such variants of NP-complete problems holds significant importance in computational complexity theory [29].

The remainder of this paper is structured as follows. Section II introduces some definitions and notations regarding Suguru puzzles' data structure and mathematical representation, as well as some relevant theoretical results. Section III presents an algorithm that verifies a solution to a Suguru puzzle of size $m \times n$ in $O(mn)$ time. Section IV discusses our proposed backtracking algorithm—which incorporates pruning optimizations—for solving arbitrary $m \times n$ Suguru puzzles. Specifically, we prove that our backtracking algorithm can solve an arbitrary $m \times n$ Suguru instance with R regions and H hint cells in $O(R \cdot (mn - H + 2)!)$ time. Section V investigates a tractable variant of the puzzle. We show that any $m \times n$ Suguru instances where $m = 1$ or $n = 1$ are solvable in linear time. Nevertheless, we argue that discovering all solutions to a

Butrahandisya was an undergraduate student at Computing Laboratory, School of Computing, Telkom University, Bandung 40257, Indonesia, email butrahandisya@gmail.com.

Muhammad Atzaki and Gia Septiana Wulandari are with Computing Laboratory, Telkom University, Bandung 40257, Indonesia, email: arzaki@telkomuniversity.ac.id, giaseptiana@telkomuniversity.ac.id

Manuscript received June 20, 2023; accepted March 21, 2024.

tractable Suguru puzzle may require a non-polynomial number of computational steps. The experimental results to evaluate our proposed algorithm's practical performance are discussed in Section VI. Lastly, the important results of this paper are summarized and concluded in Section VII.

II. Preliminaries

All arrays in this paper use one-based indexing and are written using uppercase letters. The notation A_i for a one-dimensional array A of length or size n denotes the i-th entry of array A where $1 \leq i \leq n$. The notation $A_{i,j}$ for a two-dimensional array A of m rows and n columns denotes the entry in i-th row and j-th column of array A where $1 \leq i \leq m$ and $1 \leq j \leq n$.

A. Definition and Representation of Suguru Puzzles

Before delving into the technical details and algorithms related to the Suguru puzzle, we introduce the definition of Suguru instance, configuration, and solution in Definition 1. Moreover, Fig. 1a and Fig. 1b depict a Suguru instance and one of its corresponding solutions.

Definition 1. A Suguru *instance* of size $m \times n$ is defined as a grid of m rows and n columns containing mn cells divided into one or more regions. A region is a collection of one or more orthogonally connected cells. The size of a region is the number of cells within such a region. Initially, each cell may either be empty or contain an integer between 1 and s (inclusive), where s is the size of the region it belongs to. A *configuration* of a Suguru instance is obtained by filling all of its empty cells with non-negative integers. A *solution* to a Suguru instance is a configuration that satisfies the following rules of Suguru:

1) no two cells in the same region can contain the same integer;
2) no two adjacent cells, either orthogonally or diagonally, can contain the same integer; and
3) all cells must be filled with an integer between one and the size of the region it belongs to.

In this paper, we formally represent a Suguru instance of size $m \times n$ using a two-dimensional array G of the same size such that $G_{i,j} = (h_{i,j}, r_{i,j})$ where $h_{i,j}$ and $r_{i,j}$ respectively denote the hint for the cell (i, j) and the region label to which the cell (i, j) belongs. The value $r_{i,j}$ is a positive integer between 1 and R (inclusive), where R is the number of regions in the corresponding instance. The regions are numbered using the row-major order format, i.e., the first region in the first row is labeled with one while the last region visited is labeled with R. The value $h_{i,j}$ is a non-negative integer between 0 and $s_{r_{i,j}}$ where $s_{r_{i,j}}$ denotes the size of the region labeled $r_{i,j}$. Furthermore, $h_{i,j} = 0$ if and only if the cell (i, j) is empty. A cell (i, j) that initially contains a number (or mathematically $h_{i,j} \neq 0$) is called a *hint cell*. From Definition 1, we know that a Suguru configuration C corresponding to a Suguru instance G is obtained by imposing the value of $h_{i,j}$ to a positive integer for every cell (i, j), that is, we need to fill every empty cell with a number satisfying the constraint above.

	5	
		3

(a) An instance of a Suguru puzzle.

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

(b) An example of a solution to the instance in Fig. 1a.

```
(0,1) (0,1) (0,2)
(0,1) (5,1) (0,2)
(0,1) (0,3) (3,2)
```

(c) Array G for the Suguru instance in Fig. 1a.

```
(1,1) (4,1) (1,2)
(2,1) (5,1) (2,2)
(3,1) (1,3) (3,2)
```

(d) Representation of a solution to the instance in Fig. 1a obtained by altering 0 to a positive integer in array G in Fig. 1c.

Fig. 1: An example of a Suguru instance (Fig. 1a), its solution (Fig. 1b), and data structure representations for Suguru instance and solution (Fig. 1c and Fig. 1d).

We provide an example of a Suguru instance, its solution, and its corresponding data structure formalization in Fig. 1. Using this convention, the Suguru instance and solution in Fig. 1a and Fig. 1b are respectively represented formally as two-dimensional arrays of pairs in Fig. 1c and Fig. 1d.

B. Summary of the NP-Completeness of Suguru Puzzles

Suguru puzzles are recently proven NP-complete by Robert et al. in 2022 [1]. The authors demonstrated the NP-completeness of these puzzles by establishing a polynomial-time reduction from the Planar Circuit SAT problem to the Suguru puzzle. According to Robert et al., the Planar Circuit SAT problem is similar to the Planar SAT problem, shown NP-complete by Lichtenstein [30]. The Planar SAT problem was proven NP-complete by the reduction from the 3-Quantified Boolean Formula problem, similar to the 3-SAT problem.

In the Planar Circuit SAT problem, a planar logical circuit is given and connected solely to the logic gates responsible for computing a Boolean formula. A planar circuit is a Boolean circuit that can be drawn on a plane such that none of its wires intersect.

Robert et al. in [1] construct a polynomial-time reduction using constant-sized partial instances of Suguru puzzles, known as *gadgets*, for representing objects in the Planar Circuit SAT problem. The summary of the gadgets' constructions is described as follows:

1) The construction of types of cells. Four types of cells are used to formally define the gadgets: input, output, propagated, and independent cells. These cells are explained in [1, Fig. 6].
2) The construction of **true** and **false** gadgets. The **true** and **false** gadgets are constructed using a region of size 1×2 or 2×1 in a Suguru instance. It is important to note that the orientation of the cells determines whether the gadget

is either a **true** or **false** gadget. For further illustration of these gadgets, see [1, Fig. 4] and [1, Fig. 5].

3) The construction of the **not** gadget. The **not** gadget is created using a Suguru instance of size 5×11. This gadget takes an input and produces an output. The **not** gadget is illustrated in [1, Fig. 7].
4) The construction of the **or** and **and** gadgets. The **or** and **and** gadgets are constructed using a Suguru instance of size 5×11. These gadgets take two inputs and generate an output. Although the **or** and **and** gadgets are alike, they differ in some of the predetermined cells. Pictorial representations of these gadgets are available in [1, Fig. 8] and [1, Fig. 9].
5) The construction of **split** and **split stop** gadgets. The **split** gadget is created using a Suguru instance of size 5×22 and propagates its input to two outputs of identical values (**true** or **false**). In contrast to the **split** gadget, the **split stop** gadget propagates its input to just one of the outputs. The **split** and **split stop** gadgets are illustrated in [1, Fig. 10] and [1, Fig. 11], respectively.
6) The construction of the horizontal and vertical isolator and connector. Robert et al. in [1] introduced horizontal and vertical isolators and connectors to connect all of the logical gadgets. The horizontal isolator, horizontal connector, vertical isolator, and vertical connector are respectively illustrated in [1, Fig. 12], [1, Fig. 13], [1, Fig. 14], and [1, Fig. 15].

According to Robert et al. [1], it is possible to transform any Planar Circuit SAT instance into a Suguru instance using the gadgets above. Furthermore, if the planar circuit has a maximum of x logical gadgets in a row of its layout and a maximum of y logical gadgets in a column, the corresponding Suguru instance constructed contains $x \times 22$ rows and $y \times 15 - 10$ columns. Thus, the Suguru instance constructed is polynomially proportional to the input size. To determine whether a formula in the Planar Circuit SAT instance is satisfiable, we set the output in the corresponding Suguru instance to **true**. The formula is satisfiable if a solution to the Suguru instance exists. Consequently, the polynomial-time reduction is established. Moreover, since the compliance of Suguru configuration to the puzzle's rules can be carried out in polynomial time, they belong to the NP-complete class. Section III of this paper also discusses a polynomial-time verification algorithm that can check the compliance of an arbitrary Suguru configuration to the puzzle's rules.

C. The Non-existence of a Particular 2×2 Subgrid

In a Suguru instance, it is possible to have a 2×2 subgrid that contains more than one complete region. To formally illustrate this condition, we first discuss some collections of cells in Definition 2 and Definition 3. Fig. 2 illustrates an example of a grid that is a subgrid of another larger grid and a grid that is not.

Definition 2. A contiguous orthogonal collection of cells in grid S is a *subgrid* of G if there exists S in G and the region for each cell in S is identical to the region for each cell in G.

2	5
3	1

(a) Grid S_1 of size 2×2.

2	5
3	1

(b) Grid S_2 of size 2×2.

1	4	1
2	5	2
3	1	3

(c) Grid G of size 3×3.

Fig. 2: Grid S_1 in Fig. 2a is a subgrid of grid G in Fig. 2c, while Grid S_2 in Fig. 2b is not.

Definition 3. A region is *completely inside* a subgrid if all its cells are contained within the subgrid.

In the following theorem, we prove an elementary property that a Suguru instance with a 2×2 subgrid containing more than one complete region has no solution.

Theorem 1. *If G is a Suguru instance of size $m \times n$ that contains at least one 2×2 subgrid with more than one complete region, then G has no solution.*

Proof. Suppose G is a Suguru instance that contains a 2×2 subgrid S consisting of more than one complete region, then at least two cells within S must contain the integer 1. Since all cells in S are adjacent to each other, this results in two adjacent cells having the same number, which violates one of the Suguru puzzle's rules. □

III. VERIFYING SUGURU SOLUTIONS IN POLYNOMIAL TIME

This section outlines an algorithm for verifying whether a configuration constitutes a solution to a Suguru instance. The algorithm requires a two-dimensional array of pairs of integers denoting the configuration of a particular Suguru instance.

A. Determining the Size of Each Region

Suppose we consider a Suguru configuration C containing R regions where each cell (i, j) consists of a pair $(h_{i,j}, r_{i,j})$. We define an array $S = [s_1, s_2, \ldots, s_R]$ where s_k denotes the size of region k ($1 \le k \le R$). Here, $s_k = |\{(i, j) : r_{i,j} = k\}|$. We can determine the array S by traversing all mn cells in row-major order and increment the value s_k if and only if $r_{i,j} = k$ for a cell (i, j). Thus, the asymptotic upper bound of the running time for determining the array S is $O(mn)$. Moreover, the process to construct S requires $O(R)$ space where $R \le mn$.

B. Computing the Number of Cells Containing a Particular Number within a Region

To check if every cell (i, j) has a unique value within a particular region k ($1 \leq k \leq R$), we use a list CV of "cell values" of size R whose k-th component is a list of size s_k. The notation $CV_{\alpha,\beta}$ denotes the number of cells (i, j) such that $h_{i,j} = \beta$ and $r_{i,j} = \alpha$, that is, the number of cells in region α filled with the number β. We determine CV by traversing all mn cells in row-major order and increment the value $CV_{\alpha,\beta}$ if and only if $\alpha = r_{i,j}$ and $\beta = h_{i,j}$ for a cell (i, j). The process to construct CV requires $O(mn)$ time and $O(mn)$ space.

C. Validating the Compliance of Cell Values within a Region

To verify if the value of $h_{i,j}$ in a given cell (i, j) complies with the region's size, we check the condition $1 \leq h_{i,j} \leq s_k$, where k corresponds to the region $r_{i,j}$ that the cell belongs to. The validation process for a cell takes a constant $O(1)$ time.

D. Validating the Uniqueness of Cell Values within a Region

To validate if the value of $h_{i,j}$ in a given cell (i, j) is unique within its corresponding region, we need to ensure that it satisfies the condition $CV_{\alpha,\beta} = 1$ where α and β corresponds to the region $r_{i,j}$ and the value $h_{i,j}$, respectively. The validation process for a cell takes constant $O(1)$ time.

E. Validating the Compliance of Adjacent Cell Values

This process ensures that the value of $h_{i',j'}$ of any cell (i', j') that is adjacent to the cell (i, j) differs from $h_{i,j}$. Mathematically it checks that if $|i - i'| \leq 1$ and $|j - j'| \leq 1$ for any cell $(i', j') \neq (i, j)$, then $h_{i,j} \neq h_{i',j'}$. The validation process for a cell is constant, i.e., $O(1)$, since a cell can only have at most eight adjacent cells.

F. Main Verification Process

To verify whether a configuration is a solution to a Suguru instance, we initially perform the computations outlined in Section III-A and Section III-B, which have a time complexity of $O(mn)$. Subsequently, we verify that all cell values satisfy the conditions specified in Section III-C, Section III-D, and Section III-E. This process has a time complexity of $O(mn)$, since we need to check for mn cells, and for every cell, each validation step has a constant time complexity of $O(1)$. This demonstrates that verifying whether a Suguru configuration is also a solution can be done in polynomial time. Additionally, the space complexity of this process is $O(mn)$ as the size of the list CV is proportional to mn.

IV. A BACKTRACKING APPROACH FOR SOLVING SUGURU PUZZLES

Backtracking is an algorithmic strategy used to explore all or some possible solutions to a problem by incrementally building partial solutions and testing if they satisfy the problem's constraints. The method abandons (or *prunes*) a partial solution when it determines that it cannot lead to a valid solution within the constraints. This approach involves making a series of choices and using recursion to traverse the state space tree until a solution is found or all possibilities are exhausted [31]. Backtracking is chosen as our approach for solving Suguru puzzles due to its efficiency compared to other elementary methods, such as the exhaustive search approach (see [23]–[25] for such arguments).

To solve the Suguru puzzle using a backtracking approach, we initially compute the arrays S of region sizes and CV of cell values as explained in Section III-A and Section III-B. We achieve this by incrementing s_k where $k = r_{i,j}$ for each cell (i, j) and $CV_{\alpha,\beta}$ for each hint cell (i, j), where $\alpha = r_{i,j}$ and $\beta = h_{i,j}$. Recall that CV can be considered as two-dimensional list of size R whose k-th component is a list of size s_k such that $CV_{\alpha,\beta}$ is the number of cells in region α whose hints are equal to β. These computations yield the necessary information to construct a list of length R containing sets of integers denoted by χ. Each set χ_α ($1 \leq \alpha \leq R$) within χ consists of numbers β ($1 \leq \beta \leq s_\alpha$) that do not appear in any of the initial hint cells within region α where $1 \leq \alpha \leq R$. In other words, χ_α contains a set of possible values for every empty cell in the region α. Mathematically, $\chi_\alpha = \{\beta : 1 \leq \beta \leq s_\alpha, CV_{\alpha,\beta} = 0\}$. Subsequently, the algorithm follows these steps:

1) We fill the cells in row-major order. For each hint cell (i, j), we leave it unchanged. However, for each empty cell (i, j), we attempt to fill it with the values β from the set χ_α corresponding to region $\alpha = r_{i,j}$. We iterate through the elements of χ_α in increasing order. As we fill the cell (i, j) with a value β, we simultaneously increment $CV_{\alpha,\beta}$.
2) After assigning a value β to cell (i, j), we evaluate whether backtracking is necessary based on the following conditions:
   a) any of the adjacent cells to cell (i, j) have the same value as β;
   b) there exists another cell in the same region $r_{i,j}$ with the value β (i.e., $CV_{\alpha,\beta} > 1$ where $\alpha = r_{i,j}$).

   If any of the above conditions are met, we backtrack by first undoing the value assignment of cell (i, j) and simultaneously decrementing $CV_{\alpha,\beta}$ where $\alpha = r_{i,j}$. We then proceed to try other possible values for cell (i, j). However, if no other values are available to try for cell (i, j), we continue backtracking further, moving back to the previous cell.
3) Once all cells have been successfully filled, a solution has been found, and the algorithm terminates.
4) If we backtrack after trying all possible values β for cell $(1, 1)$, or if we backtrack to the hint cell $(1, 1)$, we conclude that the instance has no solution.

To provide a more elaborate description of the algorithm, we introduce two auxiliary functions: the function NEXTCELL(i, j), which returns the next cell (i', j') following the row-major order traversal from cell (i, j), and the function MUSTBACKTRACK(i, j), which determines if backtracking is necessary from cell (i, j) based on the conditions mentioned in step 2 above.

Algorithm 1 provides a more detailed description of the

Algorithm 1 MUSTBACKTRACK(i, j) returns true if we need to backtrack based on the current condition of cell (i, j).

Input: The cell to be checked (i, j) taken from an instance $G_{i,j}$.
Output: If any of the backtracking conditions are met, the function returns true.
1: $(h, r) \leftarrow G_{i,j}$ ▷ h is the hint and r is the region label
2: $ConditionA \leftarrow$ **false**
3: **for all cell** (i', j') **adjacent to** (i, j) **do**
4: $(h', r') \leftarrow G_{i',j'}$
5: $ConditionA \leftarrow ConditionA$ **or** $h = h'$
6: **end for**
7: $ConditionB \leftarrow CV_{r,h} > 1$
8: **return** $ConditionA$ **or** $ConditionB$

MUSTBACKTRACK(i, j) function. The variable $ConditionA$ in line 2 of Algorithm 1 stores a Boolean value illustrating whether any cells adjacent to the cell (i, j) have hints equal to $h_{i,j}$. The variable $ConditionB$ in line 7 of Algorithm 1 describes the condition of whether a particular hint $h_{i,j}$ within a region r appears more than once. Notice that this algorithm involves one iteration in lines 3-5 to update the value of $ConditionA$. Since a cell (i, j) has at most eight adjacent cells, we may assume that Algorithm 1 takes $O(1)$ time for an input cell (i, j).

Algorithm 2 expounds the backtracking algorithm, utilizing NEXTCELL(i, j) and MUSTBACKTRACK(i, j) as subroutines. These functions and procedures consider the variables G and C, respectively representing the grid state during the backtracking process and the grid state that constitutes a solution. Furthermore, these processes also consider the variables CV as a list of size R whose k-th component is a list of integers of size s_k, and χ as a list of size R where each component is a set of integers. Here, R denotes the number of regions. By invoking SEARCH$(1, 1)$ outlined in Algorithm 2, the search for a solution commences recursively using the backtracking approach, starting from cell $(1, 1)$. Fig. 3 visualizes the algorithm on a 3×3 instance with two regions and four hint cells.

We use the state space tree model of the backtracking process to analyze the asymptotic complexity of the running time for Algorithm 2. This running time is measured in terms of the number of elementary operations as defined in [32]. In the following theorem, the notation $P(n, r)$ denotes the number of r-permutations of n objects, i.e., $P(n, r) = n!/(n - r)!$ where $n!$ denotes the factorial of n.

Theorem 2. *The number of elementary operations in Algorithm 2 for solving any Suguru instance of size $m \times n$ with R regions is given by $T(m, n, R)$ where*

$$T(m,n,R) \leq 1 + \sum_{\alpha=1}^{R} \left[\sum_{i=1}^{\varepsilon_\alpha} \left(\left(\prod_{\kappa=1}^{\alpha-1} \varepsilon_\kappa! \right) \cdot P(\varepsilon_\alpha, i) \cdot i \right) \right]$$

where ε_α denotes the number of empty cells in region α ($1 \leq \alpha \leq R$).

Proof. In the worst-case scenario, the number of states in the backtracking algorithm equals the number of nodes in the state space tree. In this proof, it is important to note that when we refer to a state, we specifically mean a valid state unless it is specified otherwise. By valid state, we mean a state with at least one child node within the search space tree (except for the terminal state).

Each region α allows an empty cell there to be filled with any value from the set χ_α, where χ_α is the set of numbers β ($1 \leq \beta \leq s_\alpha$) that do not appear in any of the initial hint cells within region α. Here, s_α denotes the number of cells in region α. Consequently, each region α ($1 \leq \alpha \leq R$, where R is the number of regions), contains $\varepsilon_\alpha = |\chi_\alpha|$ empty cells.

The order in which empty cells are filled does not impact the number of states in the state space tree, provided we exclude the pruning process for the condition related to the adjacent cells. This is because the number of possible values for all empty cells in a region remains identical regardless of the current state's condition. For simplicity, we consider the generated state space tree as a result of filling the empty cells in the order of region labels.

Each level in the state space tree, except for the root state (i.e., the initial puzzle state) level, corresponds to states resulting from filling a single empty cell, branching from the previous level. The immediate ε_1 levels after the root consist of states obtained by filling the empty cells in region 1. The following ε_2 levels correspond to the states obtained by filling the empty cells in region 2 after all empty cells in region 1 are filled. In general, ε_α levels starting at level $\left(\sum_{\kappa=1}^{\alpha-1} \varepsilon_\kappa\right) + 1$ until level $\sum_{\kappa=1}^{\alpha} \varepsilon_\kappa$ correspond to the states obtained by filling the empty cells in region α ($1 \leq \alpha \leq R$) after all cells in region κ for $1 \leq \kappa \leq \alpha - 1$ are filled. To visualize this state space tree, see Fig. 4.

Level 1 involves filling the first empty cell in region 1. In this level, there are ε_1 states. This can be easily understood by observing that the root state branches out into ε_1 states, as there are ε_1 possible values for the first empty cell.

Level 2 corresponds to possible states of filling the second empty cell in region 1. Notice that there are $\varepsilon_1(\varepsilon_1 - 1)$ possible states in this level because there are ε_1 states in the previous level (level 1), and each of those states branches to $\varepsilon_1 - 1$ states. Moreover, $\varepsilon_1 - 1$ possible values exist for the second empty cell, as one value is already taken to fill the first empty cell.

Generally, the number of states in the i-th level ($2 \leq i \leq \varepsilon_\alpha$) corresponding to region α can be determined based on the number of states in the preceding level. Each state from the preceding level branches to $\varepsilon_\alpha - (i - 1)$ states. This is because by the i-th level, $i - 1$ out of ε_α values in the set χ_α are no longer possible to be filled into the current empty cell, as they are already taken by $i - 1$ previous cells in the region α.

To ease our analysis, let us introduce the function $S(i, \alpha)$ to count the number of states at the i-th level within the ε_α levels corresponding to the region α. We can define it recursively as $S(i, \alpha) = S(i - 1, \alpha) \cdot (\varepsilon_\alpha - (i - 1))$, where $1 \leq i \leq \varepsilon_\alpha$, $S(1, 1) = \varepsilon_1$, and $S(1, \alpha) = S(\varepsilon_{\alpha-1}, \alpha - 1) \cdot \varepsilon_\alpha$ for any value $1 < \alpha \leq R$. The reason for this is that the preceding level

Algorithm 2 SEARCH(i, j) uses a backtracking approach by filling empty cells in row-major order to search for a solution to a Suguru instance. Executing SEARCH$(1, 1)$ commences the search for a solution starting from cell $(1, 1)$.

Input: The cell (i, j) to be filled taken from an instance $G_{i,j}$. The initial instance is represented by a two-dimensional array G.

Output: A solution to a Suguru instance (if any), or information that the instance has no solution.

```
1:  if i ≤ m then
2:      (h, r) ← G_{i,j}                        ▷ h is the hint and r is the region label of the cell (i, j)
3:      if h ≠ 0 then                           ▷ the cell (i, j) is a hint cell
4:          (i', j') ← NEXTCELL(i, j)           ▷ go to the next cell based on row-major order traversal
5:          SEARCH(i', j')
6:      else                                    ▷ the cell (i, j) is not a hint cell (i.e., it is empty)
7:          for each integer β in χ_r do
8:              G_{i,j} ← (β, r)                ▷ set β as the value for the cell (i, j)
9:              CV_{r,β} ← CV_{r,β} + 1         ▷ increment CV_{r,β} by 1
10:             if not MUSTBACKTRACK(i, j) then ▷ up to this point, filling the cell (i, j) with β does not violate the rule
11:                 (i', j') ← NEXTCELL(i, j)   ▷ go to the next cell based on row-major order traversal
12:                 SEARCH(i', j')
13:             end if
14:             G_{i,j} ← (0, r)                ▷ filling the cell (i, j) with β violates the rule, hence h is reset to 0
15:             CV_{r,β} ← CV_{r,β} − 1         ▷ decrement CV_{r,β} by 1
16:         end for
17:     end if
18: else                                        ▷ all cells in G are filled
19:     C ← G                                   ▷ set G as the configuration of the Suguru instance (which is also the solution)
20:     terminate the algorithm
21: end if
22: if (i, j) = (1, 1) then                     ▷ at this point, the instance has no solution
23:     terminate the algorithm
24: end if
```

of the first level of the region 1 is the root state level. On the other hand, for $\alpha > 1$, the preceding level of the first level corresponding to the region α is the last level from the previous region (region $\alpha - 1$). Simplifying the function for region 1, we obtain $S(i, 1) = \varepsilon_1 \cdot (\varepsilon_1 - 1) \cdot (\varepsilon_1 - 2) \cdots (\varepsilon_1 - (i - 1))$ where $1 \leq i \leq \varepsilon_\alpha$. It can be proven that $S(i, 1)$ represents the number of i-permutations of ε_1 elements. Thus, within the ε_1 levels corresponding to region 1, the i-th level has $S(i, 1) = P(\varepsilon_1, i)$ states where $1 \leq i \leq \varepsilon_1$.

Now let us consider the first level corresponding to region 2, which is level $\varepsilon_1 + 1$. Notice that $S(1, 2) = S(\varepsilon_1, 1) \cdot \varepsilon_2 = \varepsilon_1! \cdot \varepsilon_2$ states, as the preceding level (level ε_1) has $S(\varepsilon_1, 1) = P(\varepsilon_1, \varepsilon_1) = \varepsilon_1!$ states. In general, we observe that $S(i, \alpha) = S(\varepsilon_{\alpha-1}, \alpha - 1) \cdot P(\varepsilon_\alpha, i)$ for $\alpha > 1$. Simplifying the function using mathematical induction to a non-recursive form, we also observe that $S(\varepsilon_\alpha, \alpha) = S(\varepsilon_{\alpha-1}, \alpha - 1) \cdot \varepsilon_\alpha! = \varepsilon_1! \cdot \varepsilon_2! \cdot \ldots \cdot \varepsilon_\alpha!$. Therefore, the i-th level corresponding to region α has $S(i, \alpha) = \left(\prod_{\kappa=1}^{\alpha-1} \varepsilon_\kappa!\right) \cdot P(\varepsilon_\alpha, i)$ states where $1 \leq i \leq \varepsilon_\alpha$.

By considering all ε_α levels corresponding to a region α, it is clear that the number of states associated with region α is given by $\mathcal{S}_\alpha$ where

$$\mathcal{S}_\alpha = \sum_{i=1}^{\varepsilon_\alpha} \left(\left(\prod_{\kappa=1}^{\alpha-1} \varepsilon_\kappa! \right) \cdot P(\varepsilon_\alpha, i) \right).$$

Accordingly, by taking into account all levels across all regions, the maximum number of states in the tree is bounded by $\mathcal{S}$ where

$$\begin{aligned} \mathcal{S} &= 1 + \sum_{\alpha=1}^{R} \mathcal{S}_\alpha \\ &= 1 + \sum_{\alpha=1}^{R} \left[\sum_{i=1}^{\varepsilon_\alpha} \left(\left(\prod_{\kappa=1}^{\alpha-1} \varepsilon_\kappa! \right) \cdot P(\varepsilon_\alpha, i) \right) \right]. \end{aligned}$$

Furthermore, the algorithm performs i operations for each state that belongs to the i-th level of the region α where $1 \leq \alpha \leq R$. This involves branching the current state into other i invalid states, with each branching operation taking constant time. This is because the set χ_α does not shrink as we fill the cells in the region α. Thus, the number of elementary operations in Algorithm 2 is:

$$T(m, n, R) \leq 1 + \sum_{\alpha=1}^{R} \left[\sum_{i=1}^{\varepsilon_\alpha} \left(\left(\prod_{\kappa=1}^{\alpha-1} \varepsilon_\kappa! \right) \cdot P(\varepsilon_\alpha, i) \cdot i \right) \right],$$

where the inequality occurs because we omit the pruning process related to the adjacent cells. □

In an $m \times n$ Suguru instance with only one region and ε empty cells, the number of states in the corresponding state space tree as in the proof of Theorem 2 becomes $1 + \sum_{i=1}^{\varepsilon} P(\varepsilon, i)$. As a result, the number of elementary operations involved for solving this instance using Algorithm 2 becomes $T(m, n, 1) \leq 1 + \sum_{i=1}^{\varepsilon} P(\varepsilon, i) \cdot i$. Notice that

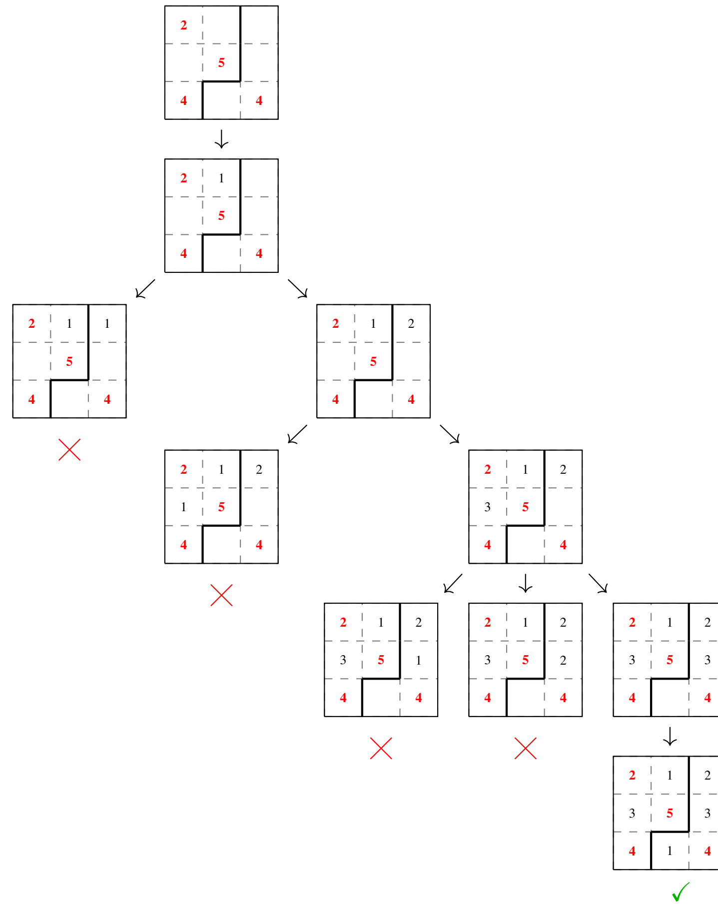


Fig. 3: Illustration of the pruned state space tree generated by Algorithm 2 for solving a 3×3 Suguru instance (depicted in the topmost part). The hint cells are indicated by bold red-colored numbers within the cells. Pruned states, which cannot possibly lead to a solution as they meet any condition specified in step 2, are indicated by red crosses. The solution found is indicated by a green check mark.

$P(\varepsilon,i) \leq \varepsilon!$ for any $1 \leq i \leq \varepsilon$, thus $\sum_{i=1}^{\varepsilon} P(\varepsilon,i) < \varepsilon \cdot \varepsilon! < (\varepsilon+1)!$. Accordingly, $1+\sum_{i=1}^{\varepsilon} P(\varepsilon,i)\cdot i < 1+\sum_{i=1}^{\varepsilon} P(\varepsilon,i)\cdot\varepsilon < 1+\varepsilon\cdot(\varepsilon+1)! < 1+(\varepsilon+2)!$. Therefore, the asymptotic upper bound for solving this instance can be written as $O((\varepsilon+2)!)$. Notice that the number of empty cells, ε, can be expressed as $mn-H$, where H is the number of hint cells in the instance. As a result, the asymptotic upper bound for solving an $m\times n$ Suguru instance with only one region and H hints using Algorithm 2 is $O((mn-H+2)!)$.

To analyze the asymptotic running time complexity of an $m\times n$ Suguru instance with R regions and H hint cells, we first prove the following lemma.

Lemma 1. *Let $a_1, a_2, \ldots, a_n$ be n positive integers, then* $(\sum_{k=1}^{n} a_k)! \geq \prod_{k=1}^{n}(a_k!)$.

Proof. From the Multinomial Theorem (see, e.g., [33], [34]), we have $\frac{(a_1+a_2+\cdots+a_n)!}{a_1!\cdot a_2!\cdots a_n!}$ is a non-negative integer. This quantity represents the number of ways to permute $\sum_{i=1}^{n} a_i$ objects with a_i indistinguishable objects of type i ($1\leq i\leq n$). Equivalently it also represents the coefficient of $x_1^{a_1}x_2^{a_2}\cdots x_n^{a_n}$ in the expansion of $(x_1+x_2+\cdots+x_n)^{(a_1+a_2+\cdots+a_n)}$. Thus $\frac{(a_1+a_2+\cdots+a_n)!}{a_1!\cdot a_2!\cdots a_n!} \geq 1$ and therefore the lemma is proven. □

The following corollary establishes the closed-form expression for the asymptotic running time of Algorithm 2 for solving a general $m\times n$ Suguru instance with H hint cells and R regions. The proof of this corollary uses the fact that the total number of empty cells in such an instance equals $mn-H$, i.e., $\sum_{\kappa=1}^{R}\varepsilon_\kappa = mn-H$.

Corollary 1. *The asymptotic running time complexity of the backtracking algorithm described in Algorithm 2 for solving $m\times n$ Suguru instance containing H hint cells and R regions is bounded above by $O(R\cdot(mn-H+2)!)$.*

Proof. From Theorem 2, the number of elementary operations of Algorithm 2, denoted by $T(m,n,R)$, satisfies the following expression

$$T(m,n,R) \leq 1+\sum_{\alpha=1}^{R}\left[\sum_{i=1}^{\varepsilon_\alpha}\left(\left(\prod_{\kappa=1}^{\alpha-1}\varepsilon_\kappa!\right)\cdot P(\varepsilon_\alpha,i)\cdot i\right)\right]. \tag{1}$$

To analyze the asymptotic upper bound of $T(m,n,R)$, let us first observe the upper bound of the expression $\sum_{i=1}^{\varepsilon_\alpha}\left[\left(\prod_{\kappa=1}^{\alpha-1}\varepsilon_\kappa!\right)\cdot P(\varepsilon_\alpha,i)\right]$ on the right hand side of (1). Notice that this expression is identical to $\mathcal{S}_\alpha$, i.e., the number

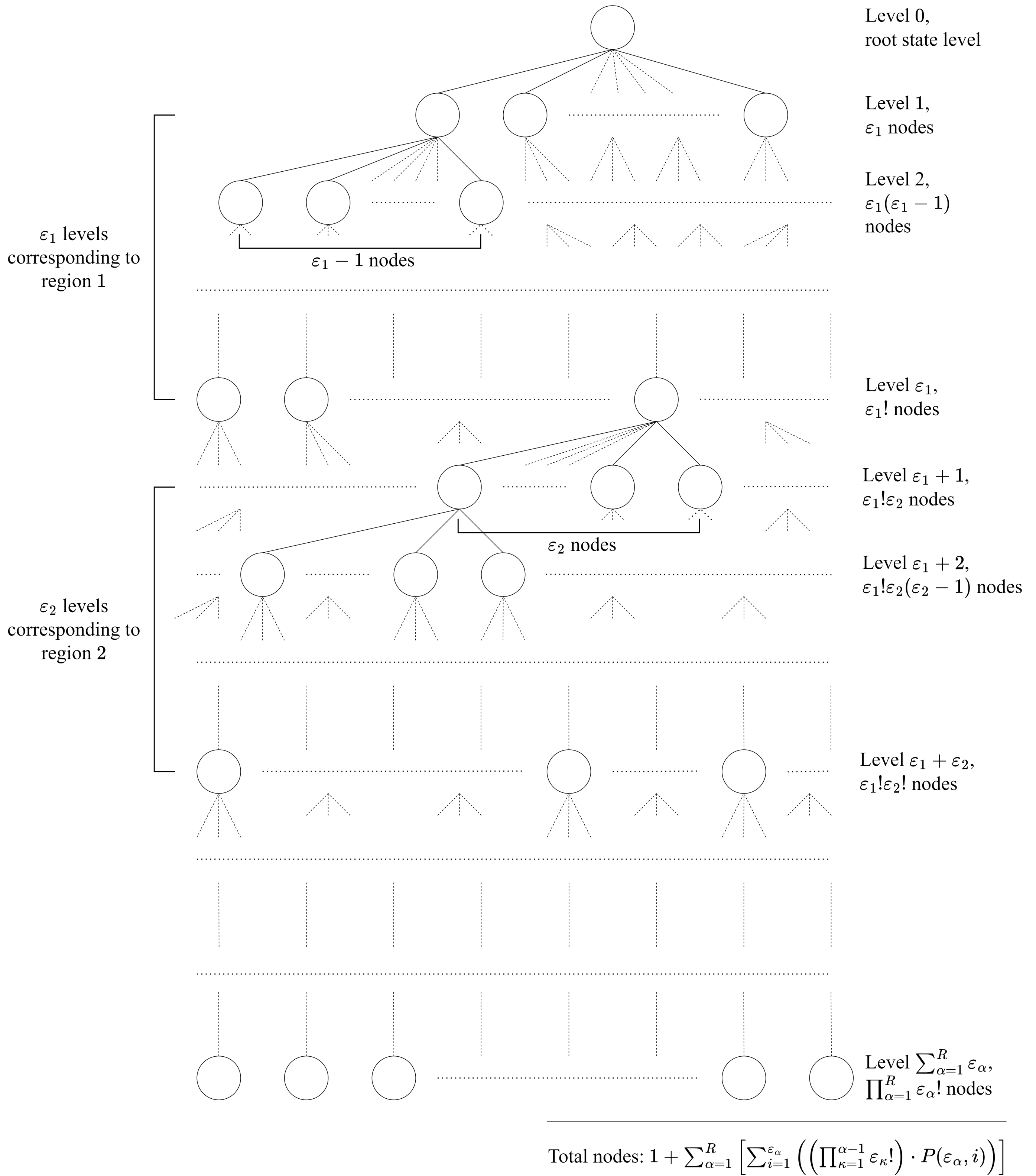


Fig. 4: The state space tree generated by Algorithm 2 for solving a general $m \times n$ Suguru puzzle with R regions. Although Algorithm 2 fills the empty cells in row-major order, the illustration presents a scenario where the empty cells are filled according to the order of region labels. This approach is discussed in the proof of Theorem 2. Each node in the tree represents a grid state, with the root node representing the initial grid state.

of states associated with region α, as defined earlier in the proof of Theorem 2. By using the fact that $P(\varepsilon_\alpha, i) \leq \varepsilon_\alpha!$, the definition of product, and Lemma 1, we observe the following chains of inequalities

$$\begin{aligned}
&\sum_{i=1}^{\varepsilon_\alpha}\left[\left(\prod_{\kappa=1}^{\alpha-1}\varepsilon_\kappa!\right)\cdot P(\varepsilon_\alpha,i)\right]\\
&\leq \sum_{i=1}^{\varepsilon_\alpha}\left[\left(\prod_{\kappa=1}^{\alpha-1}\varepsilon_\kappa!\right)\cdot \varepsilon_\alpha!\right] \text{ (because } P(\varepsilon_\alpha,i)\leq\varepsilon_\alpha!\text{)}\\
&= \sum_{i=1}^{\varepsilon_\alpha}\left[\left(\prod_{\kappa=1}^{\alpha}\varepsilon_\kappa!\right)\right] \text{ (definition of product)}\\
&\leq \sum_{i=1}^{\varepsilon_\alpha}\left(\left(\sum_{\kappa=1}^{\alpha}\varepsilon_\kappa\right)!\right) \text{ (by Lemma 1)}\\
&= \left(\sum_{\kappa=1}^{\alpha}\varepsilon_\kappa\right)!\cdot\sum_{i=1}^{\varepsilon_\alpha}1.
\end{aligned}$$

Thus, we have

$$\sum_{i=1}^{\varepsilon_\alpha}\left[\left(\prod_{\kappa=1}^{\alpha-1}\varepsilon_\kappa!\right)\cdot P(\varepsilon_\alpha,i)\cdot i\right] \leq \left(\sum_{\kappa=1}^{\alpha}\varepsilon_\kappa\right)!\cdot\sum_{i=1}^{\varepsilon_\alpha}i.$$

Accordingly, we can bound the right hand side of (1) as follows

$$\begin{aligned}
&\sum_{\alpha=1}^{R}\left(\sum_{i=1}^{\varepsilon_\alpha}\left[\left(\prod_{\kappa=1}^{\alpha-1}\varepsilon_\kappa!\right)\cdot P(\varepsilon_\alpha,i)\cdot i\right]\right)\\
&\leq \sum_{\alpha=1}^{R}\left(\left(\sum_{\kappa=1}^{\alpha}\varepsilon_\kappa\right)!\cdot\sum_{i=1}^{\varepsilon_\alpha}i\right)\\
&< \sum_{\alpha=1}^{R}\left(\left(\sum_{\kappa=1}^{\alpha}\varepsilon_\kappa\right)!\cdot{\varepsilon_\alpha}^2\right)\\
&< \sum_{\alpha=1}^{R}\left(\left(\sum_{\kappa=1}^{R}\varepsilon_\kappa\right)!\cdot{\varepsilon_\alpha}^2\right)\\
&= \sum_{\alpha=1}^{R}\left((mn-H)!\cdot{\varepsilon_\alpha}^2\right)\\
&< \sum_{\alpha=1}^{R}\left((mn-H)!\cdot(mn-H)^2\right) \text{ (since } \varepsilon_\alpha < mn-H\text{)}\\
&= R\cdot(mn-H)!\cdot(mn-H)^2\\
&< R\cdot(mn-H)!\cdot(mn-H+1)\cdot(mn-H+2)\\
&= R\cdot(mn-H+2)!.
\end{aligned}$$

Therefore, the corollary is proven. □

Notice that Corollary 1 complies with the condition when the instance contains only one region, namely, the expression $O(R\cdot(mn-H+2)!)$ becomes $O((mn-H+2)!)$. Unsurprisingly, this corollary aligns with a factorial complexity class, considering the n-queen problem, another NP-complete problem that shows factorial complexity when solved using the backtracking technique [31].

V. Tractable Variants of Suguru Puzzles

There are occasions where NP-complete problems, known for their computational hardness, have solvable subproblems in polynomial time. This means that although the general problem remains NP-complete, certain sub-problems within it can be efficiently solved. One recent example of such a case is the Yin-Yang puzzle, which, despite being NP-complete, possesses a polynomial-time solution for the puzzles of size $m\times n$ where $m\leq 2$ or $n\leq 2$, including the search of all possible solutions [21, Theorem 2 and Theorem 3]. Another case involves Nonogram, another NP-complete puzzle that has a tractable variant. Specifically, the subproblem where each row or column consists of a single block of connected cells can be solved in polynomial time by transforming it into a 2-SAT problem [35]. Several tractable problems are also found in computational graph theory. The existence of the *maximum clique* in a general graph is NP-complete, but the problem can be solved in polynomial time if the graph is planar [36]. Moreover, determining the *minimum set cover* and the *maximum independent set* in bipartite graphs are tractable even though these problems are NP-complete in arbitrary graphs [36]. These tractable subproblems provide some ways to identify specific situations within NP-complete problems where efficient solutions are attainable. By isolating these tractable subproblems, we may gain valuable insights into the underlying structure and properties of the NP-complete problems.

This section demonstrates that Suguru puzzles of size $1\times n$ and $m\times 1$ can be solved in polynomial time. To ease our analysis, we focus on the instance of size $1\times n$, noting that the case of size $m\times 1$ can be transformed into this form. A $1\times n$ Suguru puzzle is a variant of the Suguru puzzle, where the grid is restricted to a single row of length n. Two key characteristics of a $1\times n$ Suguru puzzle are: 1) the shapes of all regions are rectangular; 2) each cell can be adjacent to at most two other cells.

Before delving into the general solution for a $1\times n$ Suguru puzzle, let us first introduce a specific variant referred to as the *hintless* $1\times n$ Suguru. This variant is characterized by the absence of hint cells. In the hintless $1\times n$ Suguru, we divide the n cells into R regions, which are sequentially labeled from left to right with integers ranging from 1 to R. Each region k consists of s_k cells. Furthermore, we represent the value in the i-th cell of the puzzle as v_i. To solve the hintless $1\times n$ instance, we use an algorithm that follows these straightforward steps:

1) For each region k ($1\leq k\leq R$), we assign the values 1 to s_k to all s_k cells in the region. Specifically, we fill the i-th cell in each region k with the value i.
2) We examine each i-th cell ($1\leq i\leq n-1$) in the grid in left-to-right order and consider two cases when $v_i = v_{i+1}$:
   a) If $s_\alpha = 1$, where α represents the region label of the $(i+1)$-th cell in the puzzle, we conclude that the instance has no solution.
   b) Otherwise, we swap the values between v_{i+1} and v_{i+2}.

The aforementioned algorithm clearly runs in $O(n)$ time. The following example outlines the step-by-step construction

1	1	2	1	2	3

(a) Instance G after filling each region's i-th cell with i.

1	2	1	1	2	3

(b) Instance G after swapping the values of v_2 and v_3.

1	2	1	2	1	3

(c) Instance G after swapping the values of v_4 and v_5.

Fig. 5: Solving a hintless 1×6 Suguru instance G with three regions.

of a hintless 1×6 Suguru puzzle.

Example 1. Consider a hintless 1×6 Suguru puzzle instance G, where the cells are divided into three regions: region 1 consists of the first cell, region 2 consists of the next two cells, and region 3 consists of the remaining three cells. To obtain a solution, we begin by sequentially filling the i-th cell within each region with the value i. As a result, the current values for v_1, v_2, v_3, v_4, v_5, and v_6 are respectively set as $1, 1, 2, 1, 2,$ and 3 (see Fig. 5a). Currently, v_1 is equal to v_2, and considering $s_2 > 1$, we proceed by swapping the values of v_2 and v_3 (see Fig. 5b). Consequently, v_3 is now equal to v_4, and since $s_3 > 1$, we perform another swap between v_4 and v_5 (see Fig. 5c). Ultimately, the resulting values $v_1 = 1$, $v_2 = 2$, $v_3 = 1$, $v_4 = 2$, $v_5 = 1$, and $v_6 = 3$ form a solution. This process is visually summarized in Fig. 5.

The aforementioned algorithm is specifically designed to solve a hintless $1 \times n$ Suguru puzzle. It cannot solve a general $1 \times n$ Suguru puzzle due to the presence of initial hint cells. The reason is that the swapping process described in the algorithm cannot be applied to initial hint cells since they are not allowed to be modified.

We discuss the algorithm to solve a general $1 \times n$ Suguru puzzle in the proof of the following theorem, which is refined from the explanation in [37].

Theorem 3. *The construction of a solution (or determination of its non-existence) to any instance of a $1 \times n$ Suguru puzzle can be achieved in $O(n)$.*

Proof. Suppose we have a Suguru puzzle of size $1 \times n$, where the cells are divided into R regions with each region labeled in left-to-right order with integers from 1 to R. Each region k contains s_k cells, and each cell in region k is labeled $c_{k,i}$ indicating that it is the i-th cell in region k in left-to-right order where $1 \leq i \leq s_k$. Moreover, the value presents in cell $c_{k,i}$ is denoted as $v_{k,i}$. We first check if any adjacent cells share the same number to ensure a valid initial condition. If we find such a pair of cells, we determine that the instance has no solution. Subsequently, for each region k, we define the sets α_k, β_k, and χ_k: the set $\alpha_k = \{1, 2, \dots, s_k\}$ contains the possible numbers that can appear in any cell of region k; the set β_k contains the numbers in the initial hint cells in region k; the set $\chi_k = \alpha_k \setminus \beta_k$ contains the numbers that can be used to fill the non-hint cells in region k. By utilizing an efficient implementation of the set data structure, the initial values of α_k, β_k, and χ_k for all k ($1 \leq k \leq R$) can be determined in linear time complexity $O(n)$ as each set operation such as inserting, removing, or finding an element can be performed in constant time $O(1)$. Moreover, we define the set ρ_k for each region k representing the set of numbers that the right-most cell of the region k (i.e., cell c_{k,s_k}) can only be filled with, considering only the condition of the region $k-1$ and ignoring the condition of the region $k+1$. We set ρ_1 initially as follows: if the right-most cell in region 1 (i.e., cell c_{1,s_1}) is a hint cell, we set $\rho_1 = \{v_{1,s_1}\}$ (i.e., the set containing only the number in the right-most cell of region 1); otherwise, we set $\rho_1 = \chi_1$. Subsequently, we proceed to determine $\rho_2, \rho_3, \dots, \rho_R$ by going through each region k ($2 \leq k \leq R$). We perform the following steps for each region:

1) We check if $|\chi_k| = 1$, in which case there is only one way to fill the empty cell in the region k.
2) If $|\rho_{k-1}| = 1$ and $v_{k,1} \in \rho_{k-1}$, we determine that there is no possible solution for the instance. This is because this condition implies that we have two adjacent cells with the same value.
3) If $|\rho_{k-1}| = 1$ and $|\chi_k| = 2$, then there are two ways to fill the empty cells in region k, and one of these ways may conflict with $v_{k-1,s_{k-1}}$ (i.e., the number present in right-most cell in the previous region, $k-1$). For each way, we verify if the condition $v_{k,1} \notin \rho_{k-1}$ is satisfied. If the condition holds for such a way, we call it a valid way. For each valid way, we add v_{k,s_k} to ρ_k.
4) If the previous step does not hold, we set ρ_k as follows: if the right-most cell in region k is a hint cell, we set $\rho_k = \{v_{k,s_k}\}$ (i.e., the set containing only the number in the right-most cell of region k); otherwise, we set $\rho_k = \chi_k$.

The absence of a specific condition for when $|\rho_{k-1}| = 1$ and $|\chi_k| > 2$ can be derived by the fact that all values in χ_k is possible to be filled in the right-most cell of region k (the cell c_{k,s_k}). Specifically, when filling c_{k,s_k} with any value λ from the set χ_k where $|\chi_k| > 2$, the cardinality of χ_k minus the set consisting of λ and ρ_{k-1} where $|\rho_{k-1}| = 1$ is greater than zero, i.e., $|\chi_k \setminus (\{\lambda\} \cup \rho_{k-1})| > 0$. Consequently, if the left-most cell in region k (cell $c_{k,1}$) is empty, we ensure that we have a way to fill $c_{k,1}$ without conflicting with the right-most cell in the previous region (cell $c_{k-1,s_{k-1}}$). This process of determining $\rho_2, \rho_3, \dots, \rho_R$ can be achieved with a linear time complexity $O(n)$ by traversing s_k cells in step 1 or step 3 for each region k. Once we successfully determine all the sets $\rho_2, \rho_3, \dots, \rho_R$, we conclude that a solution to the Suguru instance exists. Subsequently, we proceed to construct the solution by going through each region k ($1 \leq k \leq R$) in reverse order (i.e., from region R to region 1). For each region, we perform the following steps:

1) If $k > 1$ and the left-most cell in region k (i.e., $c_{k,1}$) is empty and $|\rho_{k-1}| = 1$, we fill $v_{k,1}$ with any number λ

Algorithm 3 CONSTRUCT$\chi(G)$ constructs the set χ_k for every region k ($1 \leq k \leq R$) in a one-dimensional Suguru instance G.

Input: A one-dimensional Suguru instance G containing $v_{k,i}$ for each region k and cell i where $1 \leq k \leq R$ and $1 \leq i \leq s_k$.
Output: The array χ containing the set χ_k for every region k ($1 \leq k \leq R$).

```
1: α ← [α_1, α_2, ..., α_R]
2: β ← [β_1, β_2, ..., β_R]
3: χ ← [χ_1, χ_2, ..., χ_R]
4: for k ← 1 to R do
5:     α_k ← {1, 2, ..., s_k}
6:     β_k ← {v_{k,i} : IsNotEmptyCell(c_{k,i}), 1 ≤ i ≤ s_k}
7:     ▷ β_k is the set of initial hints for region k
8:     χ_k ← α_k \ β_k
9:     ▷ χ_k is the set of possible values for region k
10: end for
11: return χ
```

in χ_k that is not in ρ_{k-1}. We then update ρ_k, and χ_k accordingly.

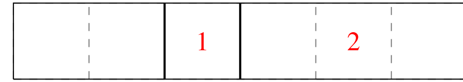
2) If the right-most cell in region k (i.e., c_{k,s_k}) is empty, we fill v_{k,s_k} with any number λ in ρ_k and update χ_k accordingly.
3) We fill all of the remaining empty cells in region k with numbers in χ_k in any order.
4) If $k > 1$, we update ρ_{k-1} by removing the number in the left-most cell of region k (i.e., $v_{k,1}$) (if any). This means that ρ_{k-1} becomes $\rho_{k-1} \setminus \{v_{k,1}\}$.

By the end of these steps, all of the cells in the Suguru instance are filled with numbers, and we have successfully constructed a solution. In step 3, the process of traversing s_k cells for each region k exhibits a time complexity of $O(n)$. This algorithm operates with a time complexity of $O(n)$ for each of its processes, leading to an overall time complexity of $O(n)$. □

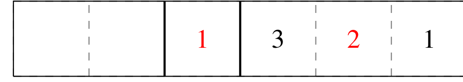
To elaborate on the algorithm described in the proof of Theorem 3, we introduce four functions:

1) ISEMPTYCELL($c_{k,i}$), which returns true if the cell $c_{k,i}$ is empty;
2) ISNOTEMPTYCELL($c_{k,i}$), which returns true if the cell $c_{k,i}$ is not empty;
3) CONSTRUCT$\chi(c)$, which returns the array χ containing sets $\chi_1, \chi_2, \ldots, \chi_R$;
4) and CONSTRUCT$\rho(c, \chi)$, which returns the array ρ containing sets $\rho_1, \rho_2, \ldots, \rho_R$.

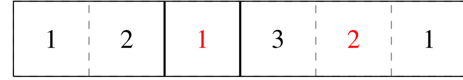
The functions CONSTRUCT$\chi(c)$ and CONSTRUCT$\rho(c, \chi)$ are described in Algorithm 3 and Algorithm 4, respectively. Furthermore, Algorithm 5 elaborates the steps described in the proof of Theorem 3, utilizing the functions ISEMPTYCELL($c_{k,i}$), ISNOTEMPTYCELL($c_{k,i}$), CONSTRUCT$\chi(c)$, and CONSTRUCT$\rho(c, \chi)$ as subroutines. These algorithms consider the variable s as an array of integers of size R, storing the size of each region.



(a) Initial instance G.



(b) Instance G after filling the cells in region 3.



(c) Instance G after filling the cells in region 1.

Fig. 6: Solving a 1×6 Suguru instance G with three regions and two hint cells.

The following example outlines the step-by-step construction of a 1×6 Suguru puzzle containing three regions with two hint cells.

Example 2. Consider a 1×6 Suguru puzzle instance G with six cells divided into three regions as in Fig. 6a. Region 1 consists of the first two cells, region 2 consists of the next cell, and region 3 consists of the remaining three cells. In this instance, the cells $c_{2,1}$ and $c_{3,2}$ are hint cells filled with the numbers 1 and 2, respectively. We first construct the sets $\chi_1 = \{1, 2\}$, $\chi_2 = \emptyset$, and $\chi_3 = \{1, 3\}$. These sets contain the numbers that do not appear in any initial hint cells of their respective regions. Subsequently, we construct the sets ρ_1, ρ_2, and ρ_3 to determine the possible values of the right-most cell of their respective regions. Since the right-most cell in region 1 (cell c_{1,s_2}) is empty, we set $\rho_1 = \chi_1$. Similarly, as the right-most cell in region 2 (cell $c_{2,1}$) is a hint cell, we set $\rho_2 = \{v_{2,1}\}$. For ρ_3, we have two ways to fill the empty cells in region 3. Using the first way, we fill cells $c_{3,1}$ and $c_{3,3}$ respectively with the numbers 1 and 3. We find that $v_{3,1} \in \rho_2$, indicates that the first way is not valid. However, using the second way, we fill cells $c_{3,1}$ and $c_{3,3}$ correspondingly with the numbers 3 and 1. This time, $v_{3,1} \notin \rho_2$, confirming that the second way is valid. Hence, we set $\rho_3 = \{v_{3,3}\}$. Since all ρ_1, ρ_2, and ρ_3 are successfully constructed, a solution exists for the Suguru instance. To construct the solution, we fill cells from the right-most region and move towards the first region. In this case, as cell $c_{3,1}$ is empty and $|\rho_2| = 1$, we fill $v_{3,1}$ with a value λ from χ_3 excluding ρ_2, which in this case is $\{3\}$. Since only one possible value exists, we set $v_{3,1} = 3$. Next, since ρ_3 contains only one value, we fill $v_{3,3} = 1$. The instance after the cells in region 3 are filled is depicted in Fig. 6b. As the only cell in region 2 is a hint cell, we leave it unchanged and update ρ_1 to $\rho_1 \setminus \{v_{2,1}\} = \{2\}$. Subsequently, as ρ_1 has only one value, we fill $v_{1,2} = 2$. Finally, we fill $v_{1,1} = 1$ as depicted in Fig. 6c. The visual representation for the overall process is given in Fig. 6.

Although the algorithm described in the proof of Theorem 3 has an $O(n)$ running time upper bound, it does not retrieve all solutions to a Suguru instance of size $1 \times n$. In contrast,

Algorithm 4 CONSTRUCT$\rho(G, \chi)$ construct the set ρ_k for every region k ($1 \leq k \leq R$) in a one-dimensional Suguru instance G.

Input: A one-dimensional Suguru instance G containing $v_{k,i}$ for each region k and cell i where $1 \leq k \leq R$ and $1 \leq i \leq s_k$, and the array χ containing the sets $\chi_1, \chi_2, \ldots, \chi_R$.

Output: The array ρ containing the sets ρ_k for every region k ($1 \leq k \leq R$), ρ_k denotes the set of possible values for the right-most cell of the region k ($1 \leq k \leq R$).

1: $\rho \leftarrow [\rho_1, \rho_2, \ldots, \rho_R]$
2: **if** ISNOTEMPTYCELL(c_{1,s_1}) **then**
3: $\quad \rho_1 \leftarrow \{v_{1,s_1}\}$ ▷ c_{1,s_1} is a hint cell
4: **else**
5: $\quad \rho_1 \leftarrow \chi_1$ ▷ ρ_1 can be filled with any value in $\chi_1 = \alpha_1 \setminus \beta_1$
6: **end if**
7: **for** $k \leftarrow 2$ to R **do**
8: $\quad$ **if** $|\chi_k| = 1$ **then**
9: $\quad\quad$ fill the one remaining empty cell in region k with the only number in χ_k
10: $\quad$ **end if**
11: $\quad$ **if** $|\rho_{k-1}| = 1$ **and** $v_{k,1} \in \rho_{k-1}$ **then** ▷ if this condition holds, we terminate as the instance has no solution
12: $\quad\quad$ **return** empty array
13: $\quad$ **end if**
14: $\quad$ **if** $|\rho_{k-1}| = 1$ **and** $|\chi_k| = 2$ **then**
15: $\quad\quad$ **if** the first way of filling cells in region k is a valid way **then**
16: $\quad\quad\quad$ fill the cells in region k with the first way
17: $\quad\quad\quad$ $\rho_k \leftarrow \rho_k \cup \{v_{k,s_k}\}$
18: $\quad\quad\quad$ undo the filling of cells in region k
19: $\quad\quad$ **end if**
20: $\quad\quad$ **if** the second way of filling cells in region k is a valid way **then**
21: $\quad\quad\quad$ fill the cells in region k with the second way
22: $\quad\quad\quad$ $\rho_k \leftarrow \rho_k \cup \{v_{k,s_k}\}$
23: $\quad\quad\quad$ undo the filling of cells in region k
24: $\quad\quad$ **end if**
25: $\quad$ **else**
26: $\quad\quad$ **if** ISNOTEMPTYCELL(c_{k,s_k}) **then**
27: $\quad\quad\quad$ $\rho_k \leftarrow \{v_{k,s_k}\}$
28: $\quad\quad$ **else**
29: $\quad\quad\quad$ $\rho_k \leftarrow \chi_k$
30: $\quad\quad$ **end if**
31: $\quad$ **end if**
32: **end for**
33: **return** ρ

the Yin-Yang puzzle has a tractable variant when considering puzzles of size $m \times n$ with $m \leq 2$ or $n \leq 2$, where the number of solutions is correspondingly bounded above by $O(n)$ or $O(m)$ [21, Theorem 2 and Theorem 3], enabling the retrieval of all solutions in polynomial time. Nonetheless, in the subsequent analysis, we demonstrate that for any arbitrary $1 \times n$ Suguru instance with H hints, the number of solutions is factorial in terms of $n - H$. It is not unexpected that this observation holds, considering that some computationally easy decision problems (with polynomial time complexities) show non-polynomial time complexities when their corresponding counting problems are considered [38].

The following theorem provides an upper bound on the number of solutions in a $1 \times n$ Suguru instance.

Theorem 4. *The number of solutions to a Suguru instance of size $1 \times n$ with H hint cells is bounded above by $O((n-H)!)$.*

Proof. Consider a Suguru instance of size $1 \times n$ containing H hint cells, divided into R regions. Each region k corresponds to the set χ_k containing the numbers not present in any hint cell within that region. Thus, each region k has $|\chi_k|$ empty cells, resulting in a total of $\sum_{k=1}^{R} |\chi_k| = n - H$ empty cells across all regions. It is easy to see that empty cells in each region k can be filled in $|\chi_k|!$ possible ways. Considering all regions k where $1 \leq k \leq R$, the total number of solutions to the instance is bounded by $\prod_{k=1}^{R} |\chi_k|!$. Using Lemma 1, we have $\prod_{k=1}^{R} |\chi_k|! \leq \left(\sum_{k=1}^{R} |\chi_k|\right)! = (n-H)!$, which is $O((n-H)!)$. □

VI. COMPUTATIONAL EXPERIMENTS AND RESULTS

This section discusses the computational experiments of our proposed backtracking algorithm and their results. We conducted the experiments using a C++ programming

Algorithm 5 SOLVEONEDIMENSIONALSUGURU(G) returns a solution to the $1 \times n$ Suguru instance, or determines that the instance has no solution.

Input: A one-dimensional Suguru instance G containing $v_{k,i}$ for each region k and cell i where $1 \leq k \leq R$ and $1 \leq i \leq s_k$.
Output: A solution to the Suguru instance (if any).

```
1:  for all cells γ in G do                      ▷ checking if there are adjacent cells that share a number
2:      if γ shares a number with its adjacent cell then
3:          return "no solution"
4:      end if
5:  end for
6:  χ ← CONSTRUCTχ(G)
7:  ρ ← CONSTRUCTρ(G, χ)
8:  if ρ is an empty array then
9:      return "no solution"
10: end if
11: for k ← R to 1 do                                                        ▷ construct the solution
12:     if k > 1 and ISEMPTYCELL(c_{k,1}) and |ρ_{k−1}| = 1 then
13:         fill v_{k,1} with any number λ ∈ χ_k \ ρ_{k−1}
14:         ρ_k ← ρ_k \ {v_{k,1}}
15:         χ_k ← χ_k \ {v_{k,1}}
16:     end if
17:     if ISEMPTYCELL(c_{k,s_k}) then
18:         fill v_{k,s_k} with any number λ ∈ ρ_k
19:         χ_k ← χ_k \ {v_{k,s_k}}
20:     end if
21:     fill all of the |χ_k| remaining empty cells in region k with the numbers in χ_k
22:     if k > 1 then
23:         ρ_{k−1} ← ρ_{k−1} \ {v_{k,1}}
24:     end if
25: end for
26: return G
```

language and g++ compiler version 12.2.0 on a 64-bit Windows 10 operating system with 16 GB of RAM and an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz. We used C++ to implement the algorithm because, according to empirical investigation, it performs relatively faster than other prevalently used programming languages [39]. Interested readers may retrieve the C++ source codes of our program, test cases for all instances, and other documents pertinent to the experiment at `https://github.com/abcqwq/suguru-backtrack`.

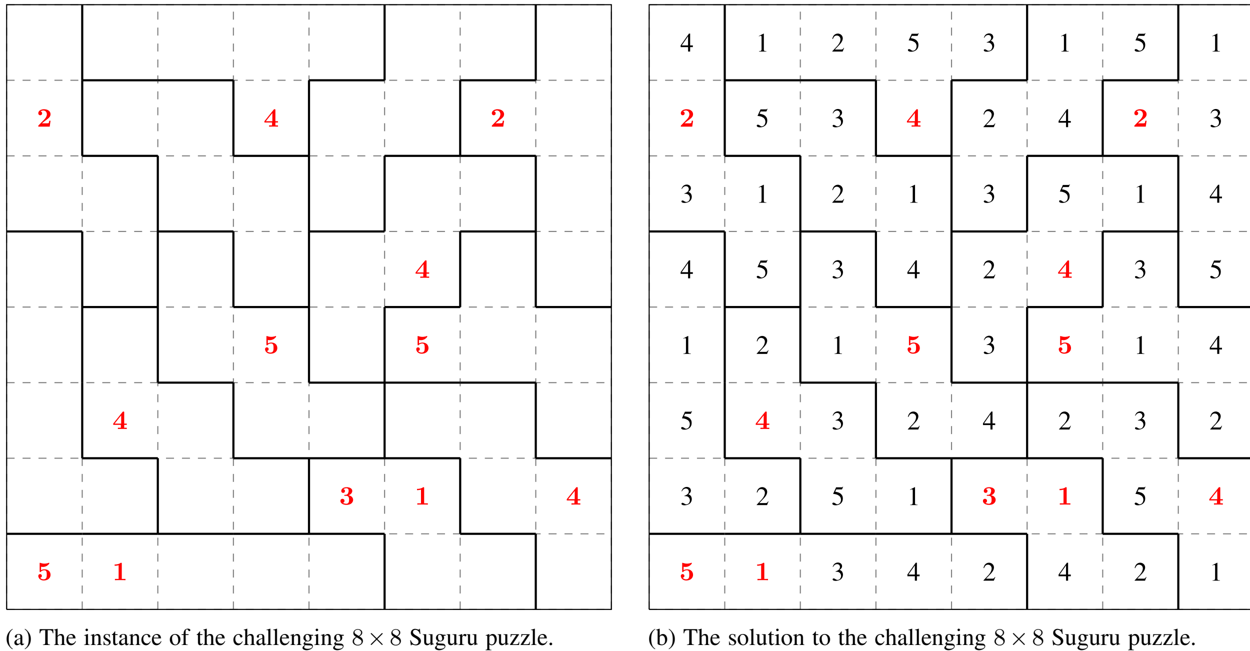
The experiment evaluated the performance of the backtracking algorithm described in Section IV, implemented in C++. To ensure comprehensive testing, we utilized a diverse set of test cases gathered from [40]. The test cases encompass a wide range of Suguru puzzles, varying in dimensions from 6×6 to 10×10. There are one hundred eighty test cases, of which around 28.9% are of size 8×8. All instances in the test cases are guaranteed to have exactly one solution. The objective of the experiment was to calculate the average running time across ten executions required for the algorithm to solve each instance.

Table 1 summarizes the algorithm's running times for solving the test cases, categorized according to their sizes. From the experiment, despite the factorial upper bound of the running time according to Corollary 1, it is evident that the C++ implementation of our proposed algorithm solves all instances in less than 0.5 seconds. Unfortunately, we could not test the implementation with larger puzzles due to the limitation of the test cases. It is important to note that the puzzle size does not solely determine the running time of the backtracking algorithm. Factors such as the number of regions, the number of hint cells, the position of the hint cells, and their arrangements also affect the algorithm's running time. As a result, two equally sized test cases may yield varying outcomes and larger test cases potentially have a shorter running time than the smaller ones.

Figure 7 illustrates a particularly challenging instance of the Suguru puzzle for the proposed algorithm, showcasing the longest running time among all test cases. Although, at first glance, the puzzle appears to be a standard 8×8 Suguru grid, its intricacies lie in the number of hints, the arrangement of regions, the configuration of these regions, and the strategic placement of hint cells. These factors contribute to the proposed algorithm encountering numerous invalid states during its exploration before ultimately converging to the correct solution state. Except for this challenging instance, on average, the proposed algorithm solved all 8×8 test cases in 2.679 milliseconds. Moreover, excluding such an instance also brings the average running time for solving all instances to 1.311 milliseconds. The challenging instance in Figure 7 is the most complex puzzle regarding the time required to solve it.

Puzzle Size	Number of Test Cases	Minimum Running Time	Maximum Running Time	Average Running Time
6×6	42	0.024	3.676	0.242
7×7	24	0.032	0.597	0.119
8×8	52	0.037	470.481	11.675
9×9	24	0.092	21.497	1.830
10×10	38	0.138	3.859	1.082

Table 1. Running times (in milliseconds) taken by the backtracking algorithm for solving one hundred eighty instances. One particular test case of size 8×8 takes 470.481 milliseconds to be solved and is depicted in Figure 7. Exclusion of such a test case brings the average running time of the algorithm for solving 8×8 test case to 2.679 milliseconds.



(a) The instance of the challenging 8×8 Suguru puzzle.

(b) The solution to the challenging 8×8 Suguru puzzle.

Fig. 7: An instance of a challenging 8×8 Suguru puzzle among the test cases and its solution.

VII. Concluding Remarks and Open Problems

We present an algorithm that verifies a Suguru puzzle solution of size $m \times n$ in $O(mn)$ time and propose a backtracking technique for solving arbitrary Suguru puzzle instance of size $m \times n$ with H hint cells and R regions in $O(R \cdot (mn - H + 2)!)$ time. Despite this factorial upper bound for the asymptotic running time, all test cases from [40] were solved using a standard personal computer in less than half a second. These test cases contain Suguru puzzles with no more than 100 cells.

With the exclusion of the single test case exhibiting the maximum running time among all 8×8 puzzles (depicted in Figure 7), the average running time for the proposed algorithm in solving 8×8 test cases notably decreased to 2.679 milliseconds. This brings the overall average time for solving any instance among the test cases to 1.311 milliseconds.

We also discuss some tractable variants of Suguru puzzles. In particular, in Theorem 3, we prove that any Suguru puzzle of size $m \times n$ where either m or n equals 1 is solvable in linear time. This finding presents a particular subproblem of the Suguru puzzles where the solution can be efficiently found. In Theorem 4, we discuss the upper bound for the number of solutions to a Suguru instance of size $1 \times n$ with H hint cells. We mathematically prove that the number of solutions to such a Suguru instance is bounded above by $O((n - H)!)$. This result provides insight into the complexity of the counting problem for a tractable Suguru puzzle. We conjecture that the complexity for such a counting problem belongs to the #P class.

We conclude by suggesting more investigations into the potential use of SAT solvers in solving Suguru puzzles. The application of SAT solvers has proven highly effective in solving other NP-complete problems, such as the n-queen problem [27], and could also be a promising approach for Suguru puzzles. Additionally, SAT solvers can be used for generating Suguru puzzles with unique solutions, opening up the possibilities to explore and gain insights into the underlying structure and properties of Suguru puzzles.

Acknowledgment

We want to thank John L. from the Computer Science Stack Exchange Community for his insight we use in Theorem 3 and Algorithm 5.

References

[1] 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. 104858, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540121001905

[2] Naoki Inaba, “number block,” http://inabapuzzle.com/honkaku/nblock.html, Jul. 2001, accessed: 2023-06-19.

[3] Z. Xu and J. Mayer, "Teaching critical thinking and problem solving skills through online puzzles and games," in *7th International Conference on Distance Learning and Web Engineering*. Citeseer, 2007, pp. 321–325.

[4] G. Kendall, A. Parkes, and K. Spoerer, "A survey of NP-complete puzzles," *ICGA Journal*, vol. 31, no. 1, pp. 13–34, 2008.

[5] E. D. Demaine, "Playing games with algorithms: Algorithmic combinatorial game theory," in *International Symposium on Mathematical Foundations of Computer Science*. Springer, 2001, pp. 18–33.

[6] E. D. Demaine, Y. Okamoto, R. Uehara, and Y. Uno, "Computational complexity and an integer programming model of Shakashaka," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 97, no. 6, pp. 1213–1219, 2014.

[7] N. Ueda and T. Nagao, "NP-completeness results for Nonogram via parsimonious reductions," Department of Computer Science, Tokyo Institute of Technology, Tech. Rep., 1996.

[8] T. Yato and T. Seta, "Complexity and completeness of finding another solution and its application to puzzles," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 86, no. 5, pp. 1052–1060, 2003.

[9] M. Holzer, A. Klein, and M. Kutrib, "On the NP-completeness of the Nurikabe pencil puzzle and variants thereof," in *Proceedings of the 3rd International Conference on FUN with Algorithms*. Citeseer, 2004, pp. 77–89.

[10] M. Holzer and O. Ruepp, "The troubles of interior design–a complexity analysis of the game Heyawake," in *International Conference on Fun with Algorithms*. Springer, 2007, pp. 198–212.

[11] D. Andersson, "Hashiwokakero is NP-complete," *Information Processing Letters*, vol. 109, no. 19, pp. 1145–1146, 2009.

[12] J. Kölker, "Kurodoko is NP-complete," *Information and Media Technologies*, vol. 7, no. 3, pp. 1000–1012, 2012.

[13] Y. Takenaga, S. Aoyagi, S. Iwata, and T. Kasai, "Shikaku and Ripple Effect are NP-complete," *Congressus Numerantium*, vol. 216, pp. 119–127, 2013.

[14] C. Iwamoto, "Yosenabe is NP-complete," *Journal of Information Processing*, vol. 22, no. 1, pp. 40–43, 2014.

[15] A. Uejima and H. Suzuki, "Fillmat is NP-complete and ASP-complete," *Journal of Information Processing*, vol. 23, no. 3, pp. 310–316, 2015.

[16] C. Iwamoto and T. Ibusuki, "Dosun-Fuwari is NP-complete," *Journal of Information Processing*, vol. 26, pp. 358–361, 2018.

[17] 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)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Farach-Colton, G. Prencipe, and R. Uehara, Eds., vol. 157. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 1:1–1:24. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12762

[18] 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. [Online]. Available: https://www.jstage.jst.go.jp/article/transinf/E103.D/3/E103.D_2019FCP0004/_pdf

[19] 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.

[20] 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. [Online]. Available: https://www.jstage.jst.go.jp/article/transinf/E105.D/3/E105.D_2021FCP0001/_pdf

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] C. Bright, J. Gerhard, I. Kotsireas, and V. Ganesh, "Effective problem solving using SAT solvers," in *Maple Conference*. Springer, 2019, pp. 205–219.

[28] A. Sbrana, L. G. B. Mirisola, N. Y. Soma, and P. A. L. de Castro, "Solving NP-Complete Akari games with deep learning," *Entertainment Computing*, p. 100580, 2023.

[29] C. Bessiere, C. Carbonnel, E. Hebrard, G. Katsirelos, and T. Walsh, "Detecting and exploiting subproblem tractability," in *IJCAI: International Joint Conference on Artificial Intelligence*, 2013, pp. 468–474.

[30] D. Lichtenstein, "Planar formulae and their uses," *SIAM journal on computing*, vol. 11, no. 2, pp. 329–343, 1982.

[31] R. Neapolitan and K. Naimipour, *Foundations of algorithms*. Jones & Bartlett Publishers, 2010.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. MIT press, 2009.

[33] D. Bolton, "The multinomial theorem," *The Mathematical Gazette*, vol. 52, no. 382, pp. 336–342, 1968.

[34] K. H. Rosen, *Discrete Mathematics and Its Applications*, 8th ed. McGraw-Hill Higher Education, 2019.

[35] S. Brunetti and A. Daurat, "An algorithm reconstructing convex lattice sets," *Theoretical computer science*, vol. 304, no. 1-3, pp. 35–57, 2003.

[36] J. I. Gunawan, "Understanding Unsolvable Problem," *Olympiad in Informatics*, vol. 10, pp. 87–98, 2016.

[37] J. L. (https://cs.stackexchange.com/users/91753/john l), "Possibly Tractable Variation of Suguru Puzzles," Computer Science Stack Exchange, uRL:https://cs.stackexchange.com/q/157307 (version: 2023-02-19). [Online]. Available: https://cs.stackexchange.com/q/157307

[38] A. Antonopoulos, E. Bakali, A. Chalki, A. Pagourtzis, P. Pantavos, and S. Zachos, "Completeness, approximability and exponential time results for counting problems with easy decision version," *Theoretical Computer Science*, vol. 915, pp. 55–73, 2022.

[39] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[40] Otto Janko, "Suguru," https://www.janko.at/Raetsel/Suguru/index.htm, Oct. 2022, accessed: 2022-10-11.