OPEN ACCESS
Indo-JC Indonesia Journal of Computing
ISSN 2460-9056
socj.telkomuniversity.ac.id/indojc

Ind. Journal on Computing
Vol. 7, Issue. 3, December 2022. pp. 53-80
doi:10.34818/indojc.2022.7.3.675

Solving Tatamibari Puzzle Using Exhaustive Search Approach

Enrico Christopher Reinhard [#1], Muhammad Arzaki [*2], Gia Septiana Wulandari [*3]

[#] *undergraduate student, Computing Laboratory*
School of Computing, Telkom University, Indonesia (40257)

[1] enricocristopher04@gmail.com

[*] *Computing Laboratory*
School of Computing, Telkom University, Indonesia (40257)

[2] arzaki@telkomuniversity.ac.id,

[3] giaseptiana@telkomuniversity.ac.id

Abstract

Tatamibari is a puzzle that was first published in 2004 and was proven to be NP-complete in 2020. However, to the best of our knowledge, algorithmic investigation of the Tatamibari puzzle is relatively new and limited. There are discussions about an approach for solving the Tatamibari puzzle using the Z3 SMT solver, but this solver requires an additional library that cannot be directly executed using standard libraries in an arbitrary imperative programming language. Hence, this paper discusses an exhaustive search approach for solving an arbitrary Tatamibari puzzle. We show that this algorithm can find all solutions to an $m \times n$ Tatamibari instance with h hints in $O(\max\{m^2n^2, h^{mn-h} \cdot hmn\})$ time. We also use this algorithm to find the number of possible Tatamibari solutions in an $m \times n$ grid for some small values of m and n.

Keywords: Asymptotic Analysis, Exhaustive Search Algorithm, NP-Complete Problems, Puzzle Solver, Tatamibari Puzzle

Abstrak

Tatamibari merupakan suatu teka teki yang pertama kali diperkenalkan pada tahun 2004 dan telah dibuktikan termasuk ke dalam permasalahan NP-*complete* pada tahun 2020. Akan tetapi, sepanjang pengetahuan kami, investigasi algoritmik terkait teka teki ini masih sangat sedikit. Terdapat pembahasan mengenai penggunaan Z3 SMT *solver* untuk menyelesaikan teka teki Tatamibari ini, namun *solver* ini menggunakan pustaka (*library*) yang tidak dapat digunakan secara langung pada sembarang bahasa pemrograman imperatif. Dalam artikel ini kami mendiskusikan algoritma pencarian menyeluruh (*exhaustive search*) untuk memecahkan teka teki Tatamibari dan menunjukkan bahwa algoritma ini dapat mencari semua solusi dari teka teki Tatamibari berukuran $m \times n$ yang memiliki h petunjuk dalam waktu $O(\max\{m^2n^2, h^{mn-h} \cdot hmn\})$. Kami juga menggunakan algoritma ini untuk mencari banyaknya solusi Tatamibari pada *grid* $m \times n$ untuk beberapa nilai m dan n yang kecil.

Kata Kunci: Algoritma Pencarian Menyeluruh, Analisis Asimtotik, Masalah NP-*complete*, Pemecah Teka Teki, Teka Teki Tatamibari

I. Introduction

TATAMIBARI is a pencil-and-paper logic puzzle popularized by Nikoli, a puzzle magazine from Japan that also popularized Sudoku. This puzzle—which was first published in 2004 [1] and inspired by Japanese tatami mats—has recently been proven to be NP-complete in 2020 [2]. The NP-completeness

Received on September, 2022. Accepted on December, 2022

proof of this puzzle establishes a *Nikoli gap*[1] of 16 years, which is one of the longest for a pencil-and-paper puzzle.

Tatamibari puzzle is a one-player pencil-and-paper game played on an $m \times n$ grid of cells where each cell is either filled with the characters $+$, $-$, or $|$, or is empty. The characters $+$, $-$, and $|$ are called *hints*. The objective of the player is to partition the grid into several rectangles satisfying the following criteria:

1) The rectangles do not overlap.
2) The union of all rectangles must cover all cells within the grid.
3) Every rectangle must contain exactly one hint (either $+$, $-$, or $|$) and satisfies the following conditions:
    a) a rectangle containing $+$ must be a square (the number of rows and columns in this rectangle are identical),
    b) a rectangle containing $-$ must have a greater width than height (the number of rows in this rectangle is less than its number of columns),
    c) a rectangle containing $|$ must have a greater height than width (the number of rows in this rectangle is more than its number of columns).
4) No rectangles share the same corner.

Puzzles are forms of entertainment that give a feeling of satisfaction to the player upon completing it [4]. Solving puzzles can help the player relax while developing visual processing and logical thinking skills. Many puzzles also contain mathematical and computational aspects with connections to combinatorial and computational problems. Numerous systematic studies have been conducted on the algorithmic aspects of one-player puzzles (see [4]–[6] for extensive bibliography). Taken from [4], there have been many puzzles that have been proven to be NP-complete, such as (in alphabetical order): Blocks World, Clickomania, Corral Puzzle, Cross Sum, Cryptarithms, Instant Insanity, KPlumber, Lemmings, Light Up, Mastermind, Minesweeper, n-Puzzle, Nurikabe (see also [7]), Pearl Puzzle, Peg Solitaire, Reflections, Rush Hour, Shanghai, Slither Link, Sokoban, Solitaire, Spiral Galaxies, Sudoku (see also [8]), and Tetris. Other one-player pencil-and-paper puzzles that have been proven to be NP-complete are: Country Road [9], Dosun-Fuwari [10], Fillmat [11], Five Cells [12], Hashiwokakero [13], Herugolf [14], Heyawake [15], Hiroimono [16], Juosan [17], Kurodoko [18], Kurotto [17], Makaro [14], Moon-or-Sun [19], Nagareru [19], Nurimeizu [19], Ripple Effect [20], Shakashaka [21], Shikaku [20], Sto-Stone [3], Tatamibari [2], Tilepaint [12], Usowan [22], Yajilin [9], Yin-Yang [23], and Yosenabe [24].

The NP-completeness of Tatamibari puzzles implies the existence of a polynomial time algorithm for verifying whether an arbitrary $m \times n$ grid is also a *Tatamibari solution*.[2] Furthermore, the NP-completeness of the puzzles also infers the existence of an exponential time algorithm for solving an arbitrary $m \times n$ Tatamibari puzzle with h hints. To our knowledge, algorithmic investigation of the Tatamibari puzzle is relatively new and limited. Adler et al. [2], [25] and Bosboom [26] briefly discuss an approach for solving the Tatamibari puzzle using the Z3 SMT solver, but the authors did not provide details regarding the steps of the algorithm as well as its explicit asymptotic upper bound. Moreover, solving Tatamibari puzzles using this solver requires an additional library that cannot be directly executed using standard libraries in an arbitrary imperative programming language.

There are several approaches to solve NP-complete puzzles, such as using the integer programming model [21], the SAT solver [27]–[31], or the SMT solver [25]. Here, we discuss the exhaustive search technique as an elementary and explicit method for solving arbitrary Tatamibari puzzles. We provide an explicit upper bound of the time complexity for finding all solutions to an arbitrary Tatamibari puzzle and also prove that the solutions can be obtained in exponential time in terms of the number of hints and the size of the puzzle.

Here, we present our systematic investigation of solving Tatamibari puzzles using an exhaustive search approach, divided into four sections. We briefly summarize the NP-completeness of Tatamibari puzzles in [2] and discuss the formal representation of Tatamibari instances, configurations, and solutions in Section

[1]According to [3], a *Nikoli gap* is the amount of time between the first publication of a Nikoli puzzle and its corresponding hardness result.

[2]A Tatamibari solution is an $m \times n$ grid configuration that satisfies the four rules of the Tatamibari puzzle, the formal definition of the Tatamibari solution is discussed in Definition 1.

II. We provide an $O(hmn)$ time algorithm for verifying whether an $m \times n$ Tatamibari configuration with h hints is also a solution in Section III. Our main result of the paper is presented in Section IV and it is shown that all solutions to an $m \times n$ Tatamibari instance with h hints can be obtained using an $O(\max\{m^2n^2, h^{mn-h} \cdot hmn\})$ algorithm. We implement our proposed algorithm in C++ programming language and summarize some of its computational results in Section V. Finally, this paper is concluded in Section VI.

II. Preliminaries

In this section, we discuss related works about the NP-completeness of the Tatamibari puzzles that have been proven in 2020 [2]. We also provide the definitions of instances, configurations, and solutions for the Tatamibari puzzles.

A. NP-Completeness of Tatamibari Puzzles

The first rigorous proof regarding the NP-completeness of the Tatamibari puzzles is written by Adler et al. in [2]. Here, the author first proved that the Tatamibari puzzle is NP-hard by a reduction from the *planar rectilinear monotone 3SAT problem*. The planar rectilinear monotone 3SAT problem was proven to be NP-hard by de Berg and Khosravi [32].

An instance of the planar rectilinear monotone 3SAT problem has a planar rectilinear drawing of the clause-variable graph. Every variable is a horizontal segment on the x-axis and every clause is a horizontal segment above or below the axis. Variables are connected to the clauses in which they appear by rectilinear edges. Fig. 1 provides a rectilinear representation of a planar monotone 3SAT instance of six variables $x_1, x_2, \ldots, x_6$.

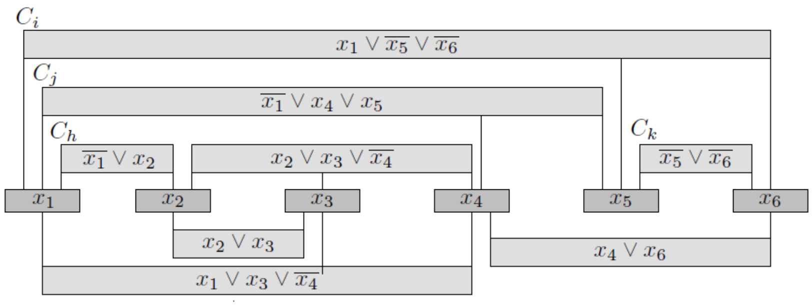

C_i
$x_1 \vee \overline{x_5} \vee \overline{x_6}$
C_j
$\overline{x_1} \vee x_4 \vee x_5$
C_h
C_k
$\overline{x_1} \vee x_2$
$x_2 \vee x_3 \vee \overline{x_4}$
$\overline{x_5} \vee \overline{x_6}$
x_1
x_2
x_3
x_4
x_5
x_6
$x_2 \vee x_3$
$x_4 \vee x_6$
$x_1 \vee x_3 \vee \overline{x_4}$

Fig. 1: A rectilinear representation of a planar monotone 3SAT instance in [32].

To prove that the Tatamibari puzzle is NP-complete, Adler et al. first showed that the puzzle is NP-hard. The authors discussed a reduction to the Tatamibari puzzle from the planar rectilinear monotone 3SAT problems. This reduction uses small partial instances of a Tatamibari puzzle, commonly referred to as *gadgets*, to simulate certain objects in the planar rectilinear monotone 3SAT problems. The author first introduced a general "gadget area hardness framework" to argue about the compilations of local gadgets whose logical representation is characterized by area coverage. There are three types of gadget construction used in the reduction, namely:

1) The construction of wire and terminator gadgets. According to [2], wire gadgets in a Tatamibari instance consist of columns containing $+$ hints surrounded by $|$ hints which encode a truth value in the parity of whether the cells are oriented with the $+$ hints in their lower left or upper left corners; while terminator gadgets are used to terminate unused wire gadgets regardless of their parity. In a planar rectilinear monotone 3SAT instance, a wire connects two different clauses separated horizontally.
2) The construction of variable gadgets. Variable gadgets in a Tatamibari instance are associated with the variable in the planar rectilinear monotone 3SAT problems, which are expressed as $x_1, x_2, \ldots$.

3) The construction of clause gadgets. Clause gadgets in a Tatamibari instance are associated with the clauses in the planar rectilinear monotone 3SAT problems, which are expressed as a disjunction of several variables or their negations, such as $x_1 \vee \overline{x}_5 \vee \overline{x}_6$.

Adler et al. constructed a reduction f that maps a planar rectilinear monotone 3SAT instance Φ to a Tatamibari instance $f(\Phi)$ using the aforementioned gadgets and show that the following conditions are satisfied:

1) if Φ consists of n variables and m clauses, then the size of $f(\Phi)$ is polynomial in $n+m$ and it can be computed in $O(p(n+m))$ time where p is a polynomial [2, Proposition 3.16];
2) an instance Φ of a rectilinear monotone 3SAT problem has a solution if and only if its corresponding Tatamibari instance $f(\Phi)$ also has a solution [2, Proposition 3.17, Proposition 3.18].

The two previously mentioned results imply that Tatamibari puzzles are NP-hard. Moreover, since any given Tatamibari solution can be checked in polynomial time (with respect to the size of the Tatamibari board and the number of hints), we infer that Tatamibari puzzles are NP-complete. We discuss an explicit polynomial time verification algorithm for checking whether arbitrary Tatamibari configurations are also Tatamibari solutions in Section III.

B. Tatamibari Instances, Configurations, and Solutions

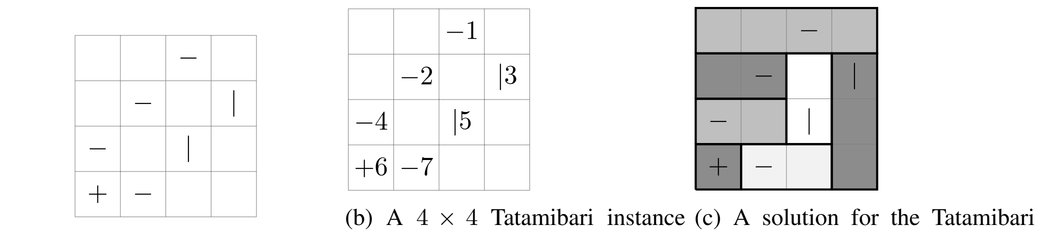
To discuss an algorithm that solves arbitrary Tatamibari instances, we first formalize the definition of a Tatamibari instance, a Tatamibari configuration, and a Tatamibari solution.

Definition 1. *An instance of a Tatamibari puzzle (or a Tatamibari instance) of size $m \times n$ is a rectangular array (or board) of m rows and n columns such that:*

1) *a cell (i, j) is the intersection between row i and column j where $1 \leq i \leq m$ and $1 \leq j \leq n$,*
2) *every cell (i, j) is either empty or is filled with precisely one character among $+$, $-$, and $|$.*

We call the characters $+$, $-$, and $|$ in a Tatamibari instance as hints. An $m \times n$ Tatamibari configuration is an $m \times n$ two-dimensional array such that every cell (i, j) is filled with a string of the form $\langle symbol \rangle \langle number \rangle$, where $\langle symbol \rangle$ is either $+$, $-$, or $|$ and $\langle number \rangle$ is an integer between 1 and h (inclusive) where h denotes the number of hints in the Tatamibari instance. The string $\langle symbol \rangle \langle number \rangle$ is abbreviated by id and is called the identity of a cell. A Tatamibari solution is a Tatamibari configuration that satisfies the four rules of the Tatamibari puzzle.

We illustrate examples of Tatamibari instances and a Tatamibari solution in Fig. 2. Fig. 2a is an example of a Tatamibari instance of size 4×4 and seven hints. For algorithmic purposes, we put a number in each hint. The hints are numbered from left to right and top to bottom fashion. The numbered Tatamibari instance in Fig. 2a is shown in Fig. 2b. One solution to the Tatamibari puzzle in Fig. 2a is given in Fig. 2c. In this solution, different regions are separated by thick lines and have different shades of colors.



(a) A 4×4 Tatamibari instance. (b) A 4×4 Tatamibari instance with numbered hints. (c) A solution for the Tatamibari puzzle in Fig. 2a.

Fig. 2: Examples of a Tatamibari instance, a Tatamibari instance with numbered hints, and a solution to a Tatamibari puzzle. Fig. 2a is a 4×4 Tatamibari instance. Fig. 2b is obtained from the instance in Fig. 2a by adding a number to every hint. Fig. 2c is a solution to the Tatamibari puzzle in Fig. 2a.

Definition 1 allows us to conveniently represents every Tatamibari instance, configuration, and solution using a two-dimensional array. An illustration regarding this representation is given in Fig. 3. In general, any Tatamibari instance can be represented using a two-dimensional array of size $m \times n$ where each entry of the array is either $*$, $+$, $-$, or $|$, where $*$ denotes an empty cell. Using this convention, the

Tatamibari instances in Fig. 2a and Fig. 2b and the Tatamibari solution in Fig. 2c can be respectively represented using the arrays in Fig. 3a, Fig. 3b, and Fig. 3c. In addition, Definition 1 also tells that every Tatamibari solution is also a Tatamibari configuration, but not vice versa. Several examples of Tatamibari configurations that are not Tatamibari solutions are given in Fig. 4.

$$\begin{array}{cccc} * & * & - & * \\ * & - & * & | \\ - & * & | & * \\ + & - & * & * \end{array}$$

(a) An array that corresponds to the Tatamibari instance depicted in Fig. 2a.

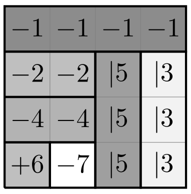
$$\begin{array}{cccc} * & * & -1 & * \\ * & -2 & * & |3 \\ -4 & * & |5 & * \\ +6 & -7 & * & * \end{array}$$

(b) An array for the Tatamibari instance with numbered hints depicted in Fig. 2b.

$$\begin{array}{cccc} -1 & -1 & -1 & -1 \\ -2 & -2 & |5 & |3 \\ -4 & -4 & |5 & |3 \\ +6 & -7 & -7 & |3 \end{array}$$

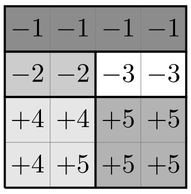
(c) An array for the Tatamibari solution depicted in Fig. 2c.

Fig. 3: Several two-dimensional array representations of a Tatamibari puzzle.



−1	−1	−1	−1
−2	−2	|5	|3
−4	−4	|5	|3
+6	−7	|5	|3

(a) The shape of the region of identity -7 is a square.



−1	−1	−1	−1
−2	−2	−3	−3
+4	+4	+5	+5
+4	+5	+5	+5

(b) There is a grid dot that is shared by four different regions.

Fig. 4: Two Tatamibari configurations which are not solutions to any Tatamibari instances.

III. Polynomial Time Algorithm for Verifying Tatamibari Solutions

We develop a polynomial time algorithm for verifying whether a Tatamibari configuration is also a Tatamibari solution. We refer to this algorithm as the Tatamibari solution verifier. This algorithm uses 0-based index convention for arrays and lists. Therefore, the top-leftmost cell in a Tatamibari board is mapped into a cell $(0, 0)$ in its two-dimensional array B representing the board. This algorithm takes an $m \times n$ two-dimensional array as input. Moreover, every cell in this array is a string of the form $\langle symbol\rangle\langle number\rangle$ called identity, abbreviated as id. In other words, the algorithm takes a Tatamibari configuration as an input. Our proposed Tatamibari solution verifier algorithm consists of three main parts, namely: a procedure to create a list of identities, a procedure to check the region of each identity, and a procedure to check the non-existence of a corner that is shared by four different regions.

A. Creating a List of Identities

Suppose we consider a Tatamibari configuration represented in an array B of size $m \times n$. We define a function Identity-List(B, m, n) to find all identities in B of size $m \times n$ as well as the information regarding the first cell and the last cell of each identity in row-major order and stores these data in a (dynamic) list L. We denote the first cell of a particular identity with (r_1, c_1) and its corresponding last cell by (r_2, c_2). This step is described in Algorithm 1.

Algorithm 1 IDENTITY-LIST(B, m, n) creates a list containing the identities as well as the first and the last cell of such identities in row-major order in a Tatamibari configuration B of size $m \times n$.

```
Input: A two-dimensional array $B$ of size $m \times n$ whose entries are of the form $\langle symbol\rangle\langle number\rangle$.
Output: A list $L$ containing triples of the form $(id, (r_1, c_1), (r_2, c_2))$ where $(r_1, c_1)$ and $(r_2, c_2)$ are correspondingly
    the first and the last cells of the identity $id$ in row-major order.
 1: $i \leftarrow 0, L \leftarrow$ an empty list
 2: while $i < m$ do
 3:     $j \leftarrow 0$
 4:     while $j < n$ do
 5:         if $B[i][j]$ is not in $L$ then
 6:             $r_1, r_2 \leftarrow i$
 7:             $c_1, c_2 \leftarrow j$
 8:             add $(B[i][j], (r_1, c_1), (r_2, c_2))$ to $L$
 9:         else
10:             update $(r_2, c_2)$ with $(i, j)$ for corresponding identity in $B[i][j]$
11:         end if
12:         $j \leftarrow j + 1$
13:     end while
14:     $i \leftarrow i + 1$
15: end while
16: return $L$
```

Observe that the maximum number of iterations in lines 2–15 of Algorithm 1 is mn. In this algorithm and many algorithms that follow, we use a dynamic list[3] L whose amortized time complexity for adding an element is constant, see [33]–[35] for details. Thus, the step to add the triple $(B[i][j], (r_1, c_1), (r_2, c_2))$ to L in line 8 takes an amortized $O(1)$ time. Therefore, the asymptotic upper bound for the running time of Algorithm 1 is $O(mn)$.

B. Checking the Region's Shape of Each Identity

Suppose we consider a Tatamibari configuration B of size $m \times n$. Every identity of a cell in this configuration has the form $\langle symbol\rangle\langle number\rangle$ where $\langle symbol\rangle$ is either $+$, $-$, or $|$, and $\langle number\rangle$ is an integer between 1 and h (inclusive) where h is the number of hints in the puzzle. The shape of each region which is a collection of the cells with the same identity must correspond to the hint symbol, that is: if the identity is $+\langle number\rangle$, then the shape of the region must be a square; if the identity is $-\langle number\rangle$, then the shape of the region must be a rectangle whose number of columns is greater than its number of rows; and if the identity is $|\langle number\rangle$, then the shape of the region must be a rectangle whose number of rows is greater than its number of columns. For example, the region which is a collection of the cells with identity -1 must be a rectangle and the number of columns in this rectangle is greater than its number of rows.

To check the shape of each region which is a collection of the cells with the same identity id, we perform the following steps. First, we determine the first and the last cells that are filled with the identity id, suppose these cells are respectively denoted by (r_1, c_1) and (r_2, c_2). The shape of the region can be determined by computing the values $\Delta r = r_2 - r_1$ and $\Delta c = c_2 - c_1$. If id corresponds to the hint symbol $+$, then $\Delta r = \Delta c$ must true; if id corresponds to the hint symbol $-$, then $\Delta r < \Delta c$ must hold; and if id corresponds to the hint symbol $|$, then $\Delta r > \Delta c$ must be satisfied. In addition, if the first and the last cells that are filled with the identity id in row-major format are respectively (r_1, c_1) and (r_2, c_2), then the identity of every cell (r, c) satisfying $r_1 \leq r \leq r_2$ and $c_1 \leq c \leq c_2$ is also id. This process is summarized in Algorithm 2. Here, the identity id is denoted using an array of length two, $id[0]$ stores the symbol while $id[1]$ stores the number. For instance, if $id = |2$, then $id[0] = |$ and $id[1] = 2$.

[3]In practice, this list can be represented using a vector data structure in C++ or a list data structure in Python.

Algorithm 2 VALID-SHAPES$(B, m, n, id, r_1, c_1, r_2, c_2)$ checks whether the shape of a region which is a collection of all cells with identity id meets the rules of Tatamibari puzzle (represented as an array B of size $m \times n$). The cell (r_1, c_1) is the top-leftmost cell with identity id while the cell (r_2, c_2) is the bottom-rightmost cell with identity id.

Input: A two-dimensional array B of size $m \times n$ whose entries are of the form $\langle symbol\rangle\langle number\rangle$, an identity value id, the cell (r_1, c_1) denoting the first cell that is filled with id, and the last cell (r_2, c_2) denoting the last cell that is filled with id. The cells are checked in row-major order.

Output: The procedure returns true if the region which is a collection of all cells with identity id satisfies the Tatamibari rules according to its hint symbol; otherwise it returns false.

```
1: $\Delta r = r_2 - r_1$; $\Delta c = c_2 - c_1$
2: if $(id[0] = -$ and $(\Delta r \geq \Delta c))$ or $(id[0] = |$ and $(\Delta r \leq \Delta c))$ or $(id[0] = +$ and $(\Delta r \neq \Delta c))$ then
3:     return false        ▷ the shape of the region does not concur with the hint symbol
4: else
5:     $i \leftarrow 0$
6:     while $i < m$ do
7:         $j \leftarrow 0$
8:         while $j < n$ do
9:             if $(i < r_1$ or $i > r_2$ or $j < c_1$ or $j > c_2)$ and $B[i][j] = id$ then
10:                return false        ▷ there is a cell with identity $id$ outside the rectangle
11:            end if
12:            if $(i \geq r_1$ and $i \leq r_2$ and $j \geq c_1$ and $j \leq c_2)$ and $B[i][j] \neq id$ then
13:                return false        ▷ there is a cell within the rectangle whose identity is different from $id$
14:            end if
15:            $j \leftarrow j + 1$
16:        end while
17:        $i \leftarrow i + 1$
18:    end while
19: end if
20: return true
```

Observe that the maximum number of iterations in lines 6–18 of Algorithm 2 is mn. Therefore, the asymptotic upper bound for the running time of Algorithm 2 is $O(mn)$.

C. Checking the Non-existence of a Corner that is Shared by Four Different Regions

One of the rules of the Tatamibari puzzles states that a corner cannot be shared by four different regions. Formally, suppose we have four adjacent cells (r, c), $(r, c + 1)$, $(r + 1, c)$, and $(r + 1, c + 1)$ within the board, where each cell respectively has the identity id_1, id_2, id_3, and id_4. This rules states that id_1, id_2, id_3, and id_4 cannot be all distinct. In other words, at least two of these four identities must be identical. We describe the algorithm to check such a condition in Algorithm 3.

Algorithm 3 VALID-CORNER(B, m, n) checks whether there is a corner that is shared by four different regions in a Tatamibari configuration represented by an array B of size $m \times n$.

Input: A two-dimensional array B of size $m \times n$ whose entries are of the form $\langle symbol\rangle\langle number\rangle$.

Output: The function returns true if no corner is shared by four different regions, otherwise it returns false.

```
1: $i \leftarrow 0$
2: while $i < m - 1$ do
3:     $j \leftarrow 0$
4:     while $j < n - 1$ do
5:         if $(B[i][j] \neq B[i+1][j])$ and $(B[i][j] \neq B[i][j+1])$ and $(B[i][j] \neq B[i+1][j+1])$ and$(B[i+1][j] \neq B[i][j+1])$ and $(B[i+1][j] \neq B[i+1][j+1])$ and $(B[i][j+1] \neq B[i+1][j+1])$ then
6:             return false        ▷ the cell $(i, j)$, $(i, j+1)$, $(i+1, j)$, and $(i+1, j+1)$ have different identities
7:         end if
8:         $j \leftarrow j + 1$
9:     end while
10:    $i \leftarrow i + 1$
11: end while
12: return true
```

Observe that the number of iterations in lines 2–11 of Algorithm 3 cannot be larger than mn. Therefore, the asymptotic upper bound for the running time of Algorithm 3 is $O(mn)$.

D. Main Verification Algorithm

We describe the main function for verifying whether a Tatamibari configuration is also a solution in Algorithm 4. This algorithm uses Algorithm 1, Algorithm 2, and Algorithm 3 as subroutines.

Algorithm 4 VERIFIER(B, m, n) checks whether an $m \times n$ array B of Tatamibari configuration is also a solution.

```
Input: A two-dimensional array $B$ of size $m \times n$ representing a Tatamibari configuration.
Output: The function returns true if $B$ is a Tatamibari solution, otherwise it returns false.
 1: $L \leftarrow$ IDENTITY-LIST$(B, m, n)$        ▷ using Algorithm 1 to find all identities in $B$
 2: $valid\_solution \leftarrow$ true
 3: $i \leftarrow 0$
 4: while ($i <$ length of $L$) and $valid\_solution$ do
 5:     $id \leftarrow$ identity of $L[i]$; $r_1 \leftarrow r_1$ of $L[i]$; $c_1 \leftarrow c_1$ of $L[i]$; $r_2 \leftarrow r_2$ of $L[i]$; $c_2 \leftarrow c_2$ of $L[i]$
 6:     $valid\_solution \leftarrow$ VALID-SHAPES$(B, m, n, id, r_1, c_1, r_2, c_2)$
 7:     ▷ checking the shape of a region with identity $id$ using Algorithm 2
 8:     $i \leftarrow i + 1$
 9: end while
10: return ($valid\_solution$ and VALID-CORNER$(B, m, n)$)
```

Notice that the asymptotic time complexity of Algorithm 4 can be determined as follows:

1) Line 1 executes the procedure FIND-IDENTITY described in Algorithm 1 whose running time is $O(mn)$.
2) Lines 4–9 is executed at most h times, where h is the number of identities. Each iteration executes the procedure VALID-SHAPES described in Algorithm 2 whose running time is $O(mn)$. Thus the asymptotic time complexity of lines 4–9 is $O(hmn)$.
3) Line 10 executes the function VALID-CORNER described in Algorithm 3 whose running time is $O(mn)$.

Therefore, assuming that $h \geq 1$, the asymptotic upper bound for the running time of Algorithm 4 is $O(mn) + O(hmn) + O(mn) = O(hmn)$.

IV. EXHAUSTIVE SEARCH TECHNIQUE FOR SOLVING TATAMIBARI PUZZLES

This section discusses an algorithm for solving any arbitrary Tatamibari instances using an exhaustive search technique. The instances are represented using two-dimensional array B of size $m \times n$ whose entries are either $+$, $-$, $|$, or $*$, where $*$ signifies an empty cell as illustrated in Fig. 3a. The main algorithm is divided into four major steps as follows:

1) a procedure to append a number to every hint,
2) a procedure to determine the possible identities (possible regions) of an empty cell,
3) a procedure to generate possible combinations of identities for all empty cells,
4) a procedure to construct a Tatamibari configuration from each of the possible combinations in step 3 and verify whether such a configuration satisfies four Tatamibari rules.

A. Appending a Number to Every Hint

Suppose we have a Tatamibari instance represented in a two-dimensional array B whose entries are either $+$, $-$, $|$, or $*$, where $*$ signifies an empty cell as illustrated in Fig. 3a. To find the solution to this Tatamibari instance, we firstly number every hint (i.e., the characters $+$, $-$, and $|$) in row-major order. The combination of a hint and its corresponding number creates an identity that is used to determine the region of an empty cell. The numbering process for every hint is performed by appending a number to each hint as described in Algorithm 5. The output of this algorithm is a Tatamibari instance with numbered hints as illustrated in Fig. 3b.

Algorithm 5 IDFIER(B, m, n) appends a unique number to every hint in an array B of a Tatamibari instance of size $m \times n$.

Input: A two-dimensional array B of a Tatamibari instance whose characters are either $+$, $-$, $|$, or $*$, where $*$, where $*$ denotes an empty cell.
Output: A two-dimensional array B of a Tatamibari instance with numbered hints. The numbering is performed in row-major order.

```
1: $x \leftarrow 0$                                         ▷ $x$ is the hint number
2: $i \leftarrow 0$
3: while $i < m$ do
4:     $j \leftarrow 0$
5:     while $j < n$ do
6:         if $B[i][j] \neq *$ then                       ▷ the cell is non-empty
7:             $x \leftarrow x + 1$
8:             $B[i][j] \leftarrow B[i][j] \parallel x$   ▷ concatenate the hint with a unique number to create a unique identity
9:         end if
10:        $j \leftarrow j + 1$
11:    end while
12:    $i \leftarrow i + 1$
13: end while
14: Return $B$
```

The purpose of Algorithm 5 is to provide a unique number to each hint. A concatenation of a hint and its number is called an identity as described in Definition 1. The maximum value of such a number also signifies the number of partitions in the solution to the corresponding Tatamibari instance (if any solution exists). Observe that the maximum number of iterations in lines 3–13 of Algorithm 5 is mn. Therefore, the asymptotic upper bound for the running time of Algorithm 5 is $O(mn)$.

B. Determining the Possible Identities of an Empty Cell

Suppose we consider an empty cell (denoted by $*$) in a Tatamibari instance represented using a two-dimensional array B. If we assume that this instance has a solution, then $*$ must be replaced by an identity (i.e., a string of the form $\langle symbol\rangle\langle number\rangle$). Observe that if the initial Tatamibari instance consists of $h > 1$ hints, then there are at most h possible identities for replacing $*$. However, we can reduce the number of possible identities that can replace $*$ using a simple observation. Notice that some identities may not be eligible to replace an empty cell. For example, consider an 8×8 Tatamibari instance with an empty cell at $(3, 3)$, the cell $(3, 6)$ with identity -3, and the cell $(3, 7)$ with identity $|4$ as depicted in Fig. 5a. It is obvious that the empty cell at $(3, 3)$ cannot have the identity $|4$ because the cell $(3, 7)$ is blocked by the cell $(3, 6)$ with identity -3.

We first describe the initial four steps to reduce the number of possible identities for an empty cell. Suppose we consider an empty cell (x, y) in a Tatamibari instance of size $m \times n$, we define the following values:

1) $minr$: the row position such that $(minr, y)$ contains an identity and the value $x - minr$ is minimum; if such an identity does not exist, then $minr = -1$;
2) $maxr$: the row position such that $(maxr, y)$ contains an identity and the value $maxr - x$ is minimum; if such an identity does not exist, then $maxr = m$;
3) $minc$: the column position such that $(x, minc)$ contains an identity and the value $y - minc$ is minimum; if such an identity does not exist, then $minc = -1$;
4) $maxc$: the column position such that $(x, maxc)$ contains an identity and the value $maxc - y$ is minimum; if such an identity does not exist, then $maxc = n$.

The value $minr$, $maxr$, $minc$, and $maxc$ provides an *orthogonal perimeter* for the possible identities of an empty cell (x, y).

For an empty cell (x, y) in a Tatamibari instance of size $m \times n$, $minr$ denotes the row position of the nearest identity located *directly above* and in the same column of the cell (x, y). The function to find the value of $minr$ given an empty cell (x, y) in a Tatamibari instance B of size $m \times n$ is described in Algorithm 6. Initially, the $minr$ is set to -1. This algorithm also uses the (dynamic) list L for storing the possible identities of an empty cell (x, y). The output of this algorithm is a pair $(minr, L)$ where L

is the list of possible identities for the empty cell.

Algorithm 6 CHECK-ABOVE(B, x, y, L) returns the row position of an identity located directly above the empty cell (x, y) in a Tatamibari instance B of size $m \times n$ as well as a list L containing such an identity. If such an identity does not exist, the algorithm returns $(-1, L)$ and L remains unchanged.

Input: A Tatamibari instance B, the location of the empty cell (x, y), and a list L containing all possible identities for the cell (x, y).

Output: The pair $(minr, L)$, here $minr$ is the row index such that $(minr, y)$ contains an identity (non-empty), is located directly above (x, y), and the value $x - minr$ is minimum, such an identity is also added to the list L; if such an identity does not exist, the algorithm returns $(-1, L)$ and L remains unchanged.

```
1: $minr \leftarrow -1$
2: $i \leftarrow x - 1$
3: while $i > -1$ do
4:     if $B[i][y] \neq *$ then
5:         $minr \leftarrow i$
6:         add $B[i][y]$ to $L$
7:         break
8:     end if
9:     $i \leftarrow i - 1$
10: end while
11: return $(minr, L)$
```

Fig. 5b provides an illustration of Algorithm 6 on the empty cell $(3, 3)$. Here, the algorithm finds $|1$ as the nearest cell located directly above $(3, 3)$ containing an identity and sets $minr = 1$ since $|1$ is located at the cell $(1, 3)$. Notice that the worst-case scenario of Algorithm 6 happens when $x = m - 1$ and all cells located above (x, y) are empty. Thus, assuming that the amortized time complexity for adding an element to L in line 6 of Algorithm 6 takes $O(1)$ time (as in the analysis of Algorithm 1), we infer that the asymptotic upper bound for the running time of Algorithm 6 is $O(m)$.

The function CHECK-BELOW is defined analogously as the previous CHECK-ABOVE function. If (x, y) is an empty cell in a Tatamibari instance of size $m \times n$, $maxr$ denotes the row position of the nearest identity located *directly below* and in the same column of the cell (x, y). The CHECK-BELOW function stores the possible identities of an empty cell (x, y) in a (dynamic) list L. This function takes a tuple (B, x, y, L) as an input and returns a pair $(maxr, L)$ such that $maxr$ is the row index of an identity located directly below the empty cell (x, y) in a Tatamibari instance B of size $m \times n$ such that $x - maxr$ is minimum. Here L is a list containing possible identities for the empty cell. Initially, the value $maxr$ is set to m. If there is no cell located directly below (x, y) containing any identity, then the function returns a pair (m, L) where L remains unchanged.

Fig. 5c illustrates CHECK-BELOW function on the empty cell $(3, 3)$. Here, the function does not find any cell containing any hint located directly below $(3, 3)$. As a result, it sets $maxr = 8$. Notice that the worst-case scenario of CHECK-BELOW function happens when $x = 0$ and all cells located below (x, y) are empty. Thus, the asymptotic upper bound for the running time of this function is $O(m)$.

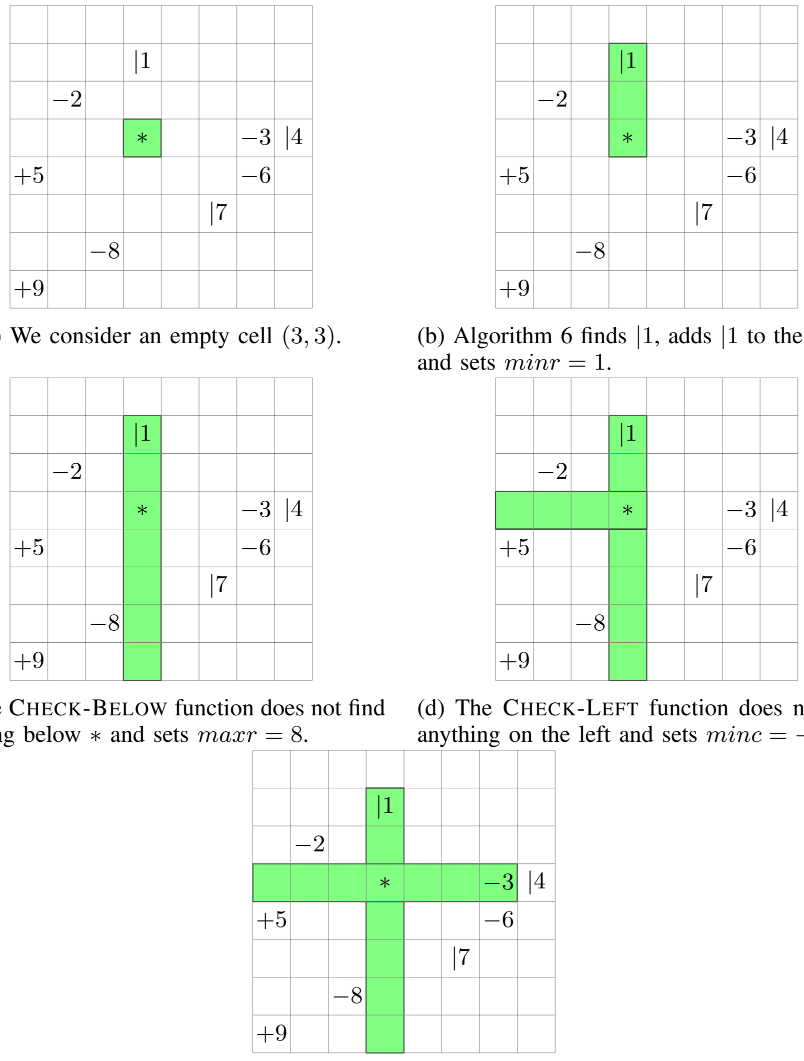
To define the leftmost perimeter of an empty cell we use the variable $minc$. Here, $minc$ signifies the column position of the nearest identity located *directly to the left* and in the same row of an empty cell (x, y) in a Tatamibari instance B of size $m \times n$. We define CHECK-LEFT as a function that takes a tuple (B, x, y, L) as an input and returns a pair $(minc, L)$ such that $y - minc$ is minimum and $(x, minc)$ contains a hint. As in the previous CHECK-ABOVE and CHECK-BELOW functions, CHECK-LEFT function uses a (dynamic) list L containing possible identities for (x, y). At first, the value $minc$ is set to 0. If there is no cell located directly to the left of (x, y) containing any identity, then the function returns a pair $(0, L)$ where L remains unchanged.

Fig. 5d depicts an illustration of CHECK-LEFT function on the empty cell $(3, 3)$. In this case, the function does not find any cell containing any hint located directly to the left of $(3, 3)$. Hence, it sets $minc = -1$. Observe that the worst-case scenario of this function happens when $y = n - 1$ and all cells located to the left of (x, y) are empty. Thus, the asymptotic upper bound for the running time of CHECK-LEFT function is $O(n)$.

To determine the rightmost perimeter of an empty cell we use the variable $maxc$. This variable represents the column position of the nearest identity located *directly to the right* and in the same row of an empty cell (x, y) in a Tatamibari instance B of size $m \times n$. Analogous to the CHECK-LEFT function, we define CHECK-RIGHT as the function that takes a tuple (B, x, y, L) and returns a pair $(maxc, L)$ such that $(x, maxc)$ contains an identity and the value $y - maxc$ is minimum. As in the previous CHECK-ABOVE, CHECK-BELOW, and CHECK-LEFT functions, this function also stores the possible identities of an empty cell (x, y) in a (dynamic) list L. Initially, the value of $maxc$ is set to n. If there is no cell located directly to the right of (x, y) with any identity, then CHECK-RIGHT function returns (n, L) where L remains unchanged.

Fig. 5e demonstrates CHECK-RIGHT function on the empty cell $(3, 3)$. The function finds the cell $(3, 6)$ with identity -3 as the nearest cell located directly to the right of $(3, 3)$ and thus it sets $maxc = 6$. It also adds -3 to the list L. Notice that the worst-case scenario of CHECK-RIGHT function happens when $y = 0$ and all cells located to the right of (x, y) are empty. Thus, the asymptotic upper bound for the running time of this function is $O(n)$.

The functions CHECK-ABOVE, CHECK-BELOW, CHECK-LEFT, and CHECK-RIGHT are independent of one another and they can be performed in any order. In Fig. 5, we illustrate the execution of these functions in sequential order.



(a) We consider an empty cell $(3, 3)$.

(b) Algorithm 6 finds $|1$, adds $|1$ to the list L, and sets $minr = 1$.

(c) The CHECK-BELOW function does not find anything below $*$ and sets $maxr = 8$.

(d) The CHECK-LEFT function does not find anything on the left and sets $minc = -1$.

(e) The CHECK-RIGHT functions finds -3, adds -3 to the list L, and sets $maxc = 6$.

Fig. 5: Visualization of the CHECK-ABOVE, CHECK-BELOW, CHECK-LEFT, and CHECK-RIGHT on cell $(3, 3)$. Here, we use 0-based indexing for the array B representing the Tatamibari board.

The values $minr$, $maxr$, $minc$, and $maxc$ provide an *orthogonal perimeter* for an empty cell (x, y). We have a rectangle whose corners are $(minr, minc)$ (top-left corner), $(minr, maxc)$ (top-right corner), $(maxr, minc)$ (bottom-left corner), and $(maxr, maxc)$ (bottom-right corner). We restrict our investigation of possible identities for (x, y) in this rectangle. Moreover, to further limit the number of possible identities for this empty cell, in general, we consider four smaller regions as follows:

1) the top-right area, i.e., all cells within the rectangle whose corners are $(minr + 1, y + 1)$ (top-left corner), $(minr + 1, maxc - 1)$ (top-right corner), $(x - 1, y + 1)$ (bottom-left corner), and $(x - 1, maxc - 1)$ (bottom-right corner);
2) the bottom-right area, i.e., all cells within the rectangle whose corners are $(x + 1, y + 1)$ (top-left corner), $(x + 1, maxc - 1)$ (top-right corner), $(maxr - 1, y + 1)$ (bottom-left corner), and $(maxr - 1, maxc - 1)$ (bottom-right corner);
3) the bottom-left area, i.e., all cells within the rectangle whose corners are $(x + 1, minc + 1)$ (top-left corner), $(x + 1, y - 1)$ (top-right corner), $(maxr - 1, minc + 1)$ (bottom-left corner), and $(maxr - 1, y - 1)$ (bottom-right corner);
4) the top-left area, i.e., all cells within the rectangle whose corners are $(minr+1, minc+1)$ (top-left corner), $(minr+1, y-1)$ (top-right corner), $(x-1, minc+1)$ (bottom-left corner), and $(x-1, y-1)$ (bottom-right corner).

We describe the function CHECK-TOP-RIGHT to find all possible identities of an empty cell (x, y) in its associated top-right area in Algorithm 7. This algorithm requires the position of an empty cell (x, y) as well as the corresponding $minr$ and $maxc$ values respectively obtained from CHECK-ABOVE and CHECK-RIGHT functions.

Algorithm 7 CHECK-TOP-RIGHT($B, x, y, minr, maxc, L$) finds all possible identities in the corresponding top-right area of an empty cell (x, y).

Input: A Tatamibari instance B, the location of the empty cell (x, y), the value $minr$ from CHECK-ABOVE function, the value $maxc$ from CHECK-RIGHT function, and a list L containing all possible identities for the cell (x, y).
Output: An updated list L containing all possible identities for the cell (x, y). The identities are obtained from the top-right area associated with (x, y).

```
1: $\ell \leftarrow maxc$
2: $i \leftarrow x - 1$
3: while $i > minr$ do
4:     $j \leftarrow y + 1$
5:     while $j < \ell$ do
6:         if $B[i][j] \neq *$ then
7:             add $B[i][j]$ to $L$
8:             $\ell \leftarrow j$
9:             break
10:        end if
11:        $j \leftarrow j + 1$
12:    end while
13:    $i \leftarrow i - 1$
14: end while
15: return $L$
```

Fig. 6 depicts an illustration of Algorithm 7 on the empty cell $(3, 3)$. The algorithm checks all possible identities in the top-right area associated with the empty cell $(3, 3)$, namely the cells $(2, 4)$ and $(2, 5)$. In general, the cells are visited in a left-to-right and bottom-up fashion. The variable ℓ is used to minimize the number of cells that must be visited in a left-to-right and bottom-up approach. Initially, ℓ is set to $maxc$, and if the algorithm finds a non-empty cell, the value ℓ is updated with the column position of this non-empty cell. Observe that the worst-case scenario of Algorithm 7 happens when $x = m - 1$, $y = 0$, $minr = -1$, $maxc = n - 1$, and all cells located in the top-right area of (x, y) are empty. Thus, assuming that the amortized time complexity for adding an element to L in line 7 of Algorithm 7 takes $O(1)$ time (as in the Analysis of Algorithm 1 and Algorithm 6), we infer that the asymptotic upper bound for the running time of Algorithm 7 is $O(mn)$.

The function CHECK-BOTTOM-RIGHT is defined analogously and it is used to find all possible identities of an empty cell (x, y) in its associated bottom-right area. This function requires the position

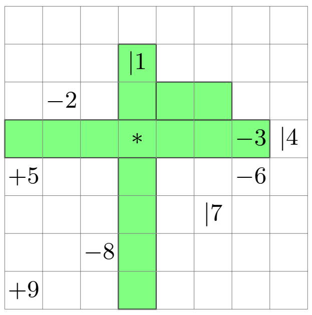


Fig. 6: Visualization of Algorithm 7 on $(3, 3)$. It checks all cells in the top-right rectangle associated with the cell $(3, 3)$ in a left-to-right and bottom-up fashion. No identities are added to the list L.

of an empty cell (x, y) as well as the corresponding $maxr$ and $maxc$ values respectively obtained from CHECK-BELOW and CHECK-RIGHT functions.

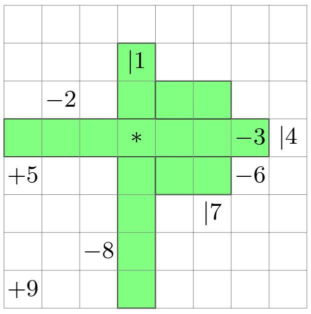
Fig. 7 illustrates the working of CHECK-BOTTOM-RIGHT function on the empty cell $(3, 3)$. This function checks all possible identities in the bottom-right area associated with the empty cell $(3, 3)$. There are four outermost iterations of this function because $maxr = 8$ and initially we inspect row 4. These iterations are consecutively illustrated in Fig. 7a, Fig. 7b, Fig. 7c, and Fig. 7d. Here, the cells are visited in a left-to-right and top-down approach. In the first iteration, the function checks cells $(4, 4)$ and $(4, 5)$ and finds no identities at such cells. In the second iteration, the function checks cells $(5, 4)$ and $(5, 5)$ and finds $|7$ at cell $(5, 5)$. Thus, the list L is added with $|7$. The third iteration only inspects cell $(6, 4)$ because any identity in cell $(6, 5)$ is block by $|7$ at cell $(5, 5)$. Similarly, the last iteration checks only the cell $(7, 4)$.

Observe that the worst-case scenario of function CHECK-BOTTOM-RIGHT happens when $x = 0$, $y = 0$, $maxr = m - 1$, $maxc = n - 1$, and all cells located in the bottom-right area of (x, y) are empty. Thus, as the time complexity of CHECK-TOP-RIGHT function, the asymptotic upper bound for the running time of this function is $O(mn)$.

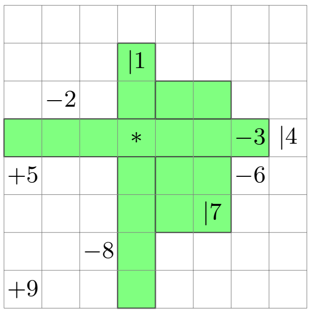
We define the function CHECK-BOTTOM-LEFT to find all possible identities of an empty cell (x, y) in its associated bottom-left area as an analog of CHECK-TOP-RIGHT and CHECK-BOTTOM-RIGHT functions. This function requires the position of an empty cell (x, y) as well as the corresponding $maxr$ and $minc$ values respectively obtained from CHECK-BELOW and CHECK-LEFT functions.

Fig. 8 depicts the working of CHECK-BOTTOM-LEFT function on the empty cell $(3, 3)$. The function checks all possible identities in the bottom-left area associated with the empty cell $(3, 3)$. The cells are visited in a right-to-left and top-down fashion. There are four outermost iterations of this function because $maxr = 8$ and initially we inspect row 4. The first, the second, and the third iterations are respectively illustrated in Fig. 8a, Fig. 8b, and Fig. 8c. In the first iteration illustrated in Fig. 8a, the function finds $+5$ in cell $(4, 0)$, adds $+5$ to L, and sets the column limit for the next iteration to 1. The second iteration illustrated in Fig. 8b checks the cells $(5, 1)$ and $(5, 2)$. Here, no identities are added to the list L. The third iteration is depicted in Fig. 8c where the function finds -8 in cell $(6, 2)$ and sets the column limit to 2. The last iteration does not check any cell. A the end of the iteration, CHECK-BOTTOM-LEFT function adds $+5$ and -8 to the list L.

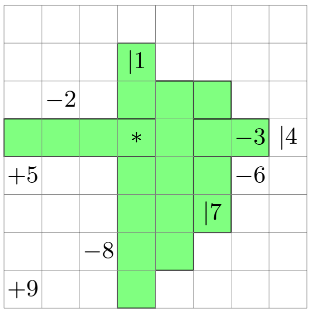
Observe that the worst-case scenario of CHECK-BOTTOM-LEFT function happens when $x = 0$, $y = n-1$, $maxr = m-1$, $minc = -1$, and all cells located in the bottom-left area of (x, y) are empty. Thus, as in the time complexity analysis of CHECK-TOP-RIGHT and CHECK-BOTTOM-RIGHT functions, the asymptotic upper bound for the running time of this function is $O(mn)$.



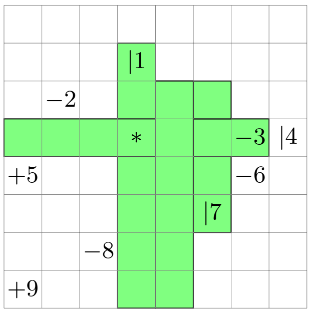
(a) Visualization of the first iteration for the outermost loop of Check-Bottom-Right function. No identities are added to the list L.



(b) Visualization of the second iteration for the outermost loop of Check-Bottom-Right function. It finds $|7$ at $(5, 5)$ and adds it to the list L.

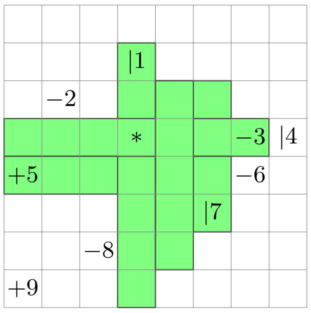


(c) Visualization of the third iteration for the outermost loop of Check-Bottom-Right function. No identities are added to the list L.

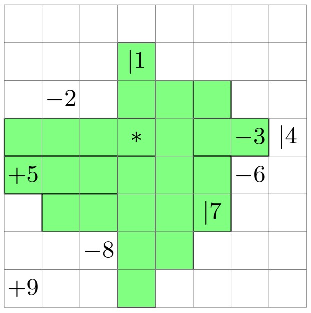


(d) Visualization of the fourth iteration for the outermost loop of Check-Bottom-Right function. No identities are added to the list L.

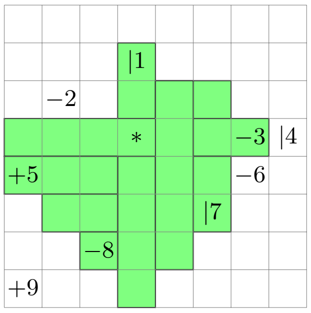
Fig. 7: Visualization of Check-Bottom-Right function on the empty cell $(3, 3)$. Here, Check-Top-Right function is completed before Check-Bottom-Right function is executed.



(a) Visualization of the first iteration for the outermost loop of Check-Bottom-Left function. It finds $+5$ and adds it to the list L. The column limit is set to 1.



(b) Visualization of the second iteration for the outermost loop of Check-Bottom-Left. No identities are added to the list L.



(c) Visualization of the third iteration for the outermost loop of Check-Bottom-Left. It finds the identity -8 and adds it to the list L. The column limit is set to 2.

Fig. 8: Visualization of Check-Bottom-Left on the empty cell $(3, 3)$. Here, Check-Top-Right and Check-Bottom-Right functions are completed before Check-Bottom-Left function is executed.

To find all possible identities of an empty cell (x, y) in its associated top-left area we define the function CHECK-TOP-LEFT which is slightly adapted from the previous CHECK-TOP-RIGHT, CHECK-BOTTOM-RIGHT, and CHECK-BOTTOM-LEFT functions. This function requires the position of an empty cell (x, y) as well as the corresponding $minr$ and $minc$ values respectively obtained from CHECK-ABOVE and CHECK-LEFT functions.

Fig. 9 depicts the working of CHECK-TOP-LEFT function on the empty cell $(3, 3)$. It checks all possible identities in the top-left area associated with the empty cell $(3, 3)$, namely the cells $(2, 0)$, $(2, 1)$, and $(2, 2)$. The cells are inspected in a right-to-left and bottom-up way. In Fig. 9, only one iteration is performed. In this iteration, the algorithm finds -2 in cell $(2, 1)$ and adds it to the list L. Observe that the worst-case scenario of CHECK-TOP-LEFT function happens when $x = m - 1$, $y = n - 1$, $minr = -1$, $minc = -1$, and all cells located in the top-left area of (x, y) are empty. Thus, as in the complexity analysis of CHECK-TOP-RIGHT, CHECK-BOTTOM-RIGHT, and CHECK-BOTTOM-LEFT functions, the asymptotic upper bound for the running time of this function is $O(mn)$.

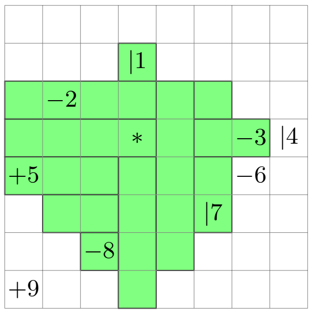


Fig. 9: Visualization of CHECK-TOP-LEFT function on the cell $(3, 3)$. It finds -2 in its only iteration of the outermost loop and adds it to the list L. Here, CHECK-TOP-RIGHT, CHECK-BOTTOM-RIGHT, and CHECK-BOTTOM-LEFT functions are completed before CHECK-TOP-LEFT function is executed.

The functions CHECK-TOP-RIGHT, CHECK-BOTTOM-RIGHT, CHECK-BOTTOM-LEFT, and CHECK-TOP-LEFT are independent of one another and they can be carried out in any order. However, all of these functions must be executed after the functions CHECK-ABOVE, CHECK-BELOW, CHECK-LEFT, and CHECK-RIGHT are completed. Moreover, to minimize the number of cells that must be checked horizontally, we use the variable ℓ as in Algorithm 7 that is updated whenever we find an identity in a row-based traversal. The process to find a list L of possible identities for an empty cell (x, y) is summarized in Algorithm 8. From previous discussion, the functions CHECK-ABOVE, CHECK-BELOW, CHECK-LEFT, and CHECK-RIGHT respectively return the tuples $(minr, L)$, $(maxr, L)$, $(minc, L)$, and $(maxc, L)$.

Algorithm 8 CHECK-POSSIBLE-REGION(B, x, y) finds all possible identities of an empty cell (x, y) using the functions CHECK-ABOVE, CHECK-BELOW, CHECK-LEFT, CHECK-RIGHT, CHECK-TOP-RIGHT, CHECK-BOTTOM-RIGHT, CHECK-BOTTOM-LEFT, and CHECK-TOP-LEFT in sequential order.

```
Input: A Tatamibari instance $B$ of size $m \times n$ and the location of the empty cell $(x, y)$.
Output: A list $L$ containing all possible identities for the cell $(x, y)$.
 1: $L \leftarrow$ an empty list
 2: $(minr, L) \leftarrow$ CHECK-ABOVE$(B, x, y, L)$
 3: $(maxr, L) \leftarrow$ CHECK-BELOW$(B, x, y, L)$
 4: $(minc, L) \leftarrow$ CHECK-LEFT$(B, x, y, L)$
 5: $(maxc, L) \leftarrow$ CHECK-RIGHT$(B, x, y, L)$
 6: $L \leftarrow$ CHECK-TOP-RIGHT$(B, x, y, minr, maxc, L)$
 7: $L \leftarrow$ CHECK-BOTTOM-RIGHT$(B, x, y, maxr, maxc, L)$
 8: $L \leftarrow$ CHECK-BOTTOM-LEFT$(B, x, y, maxr, minc, L)$
 9: $L \leftarrow$ CHECK-TOP-LEFT$(B, x, y, minr, minc, L)$
10: return $L$
```

The asymptotic upper bound for the running time of Algorithm 8 is described in Lemma 1.

Lemma 1. *The asymptotic upper bound for the running time of Algorithm 8 for an empty cell (x, y) within a Tatamibari instance B of size $m \times n$ is $O(mn)$.*

Proof. Notice that Algorithm 8 consecutively calls the procedure Check-Above,Check-Below, Check-Left, Check-Right, Check-Top-Right, Check-Bottom-Right, Check-Bottom-Left, and Check-Top-Left. From the previous analyses, we have the asymptotic upper bound for each of these algorithms is $O(m), O(m), O(n), O(n), O(mn)$, $O(mn), O(mn), O(mn)$ (resp.). Hence, with assumption that $m, n > 1$, the asymptotic upper bound for Algorithm 8 is

$$O(\max\{m, m, n, n, mn, mn, mn, mn\}) = O(mn).$$

□

C. Generating Possible Combinations of Identities for All Empty Cells

In this section, we describe a method to generate all possible combinations of identities (possible regions) for every empty cell in a Tatamibari instance. We first describe Algorithm 9 that takes an arbitrary Tatamibari instance B and returns the list E containing the positions of all empty cells in B as well as the list H such that $H[i]$ comprises all possible identities for the i-th empty cell (using 0-based index convention). Here, the lengths of E and H are identical to the number of empty cells (namely, $mn - h$).

Algorithm 9 Possible-Empty(B) returns a pair (E, H) such that E is a list containing the positions of all empty cells in B and H is a list describing the possible identities for all empty cells.

Input: A Tatamibari instance B of size $m \times n$.
Output: A pair (E, H) such that E is a list that contains the positions of all empty cells in B and H is a list such that $H[i]$ describes the possible identities of the i-th empty cell in 0-based index convention.

```
1: $E \leftarrow$ an empty list
2: $H \leftarrow$ an empty list
3: $i \leftarrow 0$
4: while $i < m$ do
5:     $j \leftarrow 0$
6:     while $j < n$ do
7:         if $B[i][j] = *$ then
8:             add $(i, j)$ to $E$                ▷ $E$ contains the locations of the empty cells in $B$
9:             add Check-Possible-Region$(B, m, n, i, j)$ to $H$
10:            ▷ $H$ is updated with the list of possible identities for cell $(i, j)$
11:        end if
12:        $j \leftarrow j + 1$
13:    end while
14:    $i \leftarrow i + 1$
15: end while
16: return $E$
17: return $H$
```

In Algorithm 9 we assume that the (amortized) worst-case complexity for adding a pair (i, j) to the list E in line 8 is $O(1)$. This is consistent with the previous analysis of Algorithm 6 and Algorithm 7. For line 9, observe that Check-Possible-Region(B, m, n, i, j) returns a list that contains at most h entries. Hence, we assume that the (amortized) worst-case complexity for adding a list of possible identities for cell (i, j) to H is $O(h)$. Observe that, according to Lemma 1, the asymptotic upper bound for the Check-Possible-Region function in line 9 is $O(mn)$. Thus the total time required to execute line 9 of Algorithm 9 that consists of Algorithm 8 and adding its return value to a list H is $O(mn) + O(h) = O(\max\{mn, h\}) = O(mn)$, since $h \leq mn$. Since the number of iterations for the innermost loop is n and the number of iterations for the outermost loop is m, the asymptotic upper bound for Algorithm 9 is $O((mn) \cdot (mn)) = O(m^2n^2)$.

We illustrate the workings of Algorithm 9 using a Tatamibari instance in Fig. 10 as follows. In Fig. 10, the indices of the empty cells in row-major order are $(0, 1)$, $(0, 2)$, $(1, 0)$, $(1, 1)$, $(1, 2)$, $(2, 0)$, and $(2, 1)$; thus we have:

$$E = [(0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1)] .$$

The list H contains the lists of possible identities for every empty cell, that is, $H[i]$ is a list of possible identities for the i-th empty cell. Using Fig. 10 as an example, the 0-th empty cell is $(0,1)$, the 1-st empty cell is $(0,2)$, and so on. Moreover, in this example, using the function CHECK-POSSSIBLE-REGION in Algorithm 8, it is clear that every empty cell can be filled with either -1 or -2. Thus, we have:

$$H = [[-1,-2],[-1,-2],[-1,-2],[-1,-2],[-1,-2],[-1,-2],[-1,-2]].$$

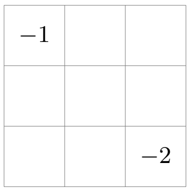


Fig. 10: An example of a 3×3 Tatamibari instance.

Notice that to store E we need to store $mn - h$ pairs of integers (i,j), thus the space requirement for storing E is $2(mn - h) = O(mn)$. The space requirement for H is analyzed as follows. Notice that $H = [H[0], H[1], \ldots, H[e-1]]$ where $e = mn - h$ denotes the number of empty cells in the board. Each $H[i]$ contains at most h entries of the form $id = \langle symbol \rangle \langle number \rangle$. Assuming that each id requires two characters, then $H[i]$ contains at most $2h$ characters. Since H contains e entries of list, the space requirement for H is bounded by $2h \cdot e = 2h(mn - h) = O(hmn)$.

D. Constructing Tatamibari Configurations from Each of Possible Combinations

In this section, we discuss a method to construct Tatamibari configurations from each of the possible combinations of identities for all empty cells. The algorithm takes the list H from Algorithm 9 as an input. Suppose we consider a Tatamibari instance of size $m \times n$ with h hints. This Tatamibari instance has $e = mn - h$ empty cells, and we have $H = [H[0], H[1], \ldots, H[e-1]]$, where $H[i]$ contains the list of possible identities for the i-th empty cell. Each $H[i]$ is at most of length h whose entries are of the form $id = \langle symbol \rangle \langle number \rangle$. Notice that, to generate a Tatamibari configuration from H, we can choose exactly one entry of each $H[i]$ where $0 \leq i \leq e-1$ and stores the chosen values in a list. We define C as the list containing lists of possible identities whose entries are obtained from choosing exactly one member of each $H[i]$ where $0 \leq i \leq e-1$. For example, if we consider the Tatamibari instance in Fig. 10, we have $H = [[-1,-2],[-1,-2],[-1,-2],[-1,-2],[-1,-2],[-1,-2],[-1,-2]]$, and thus an entry of C is a list whose components are constructed by taking exactly one member of each $H[i]$ for every $0 \leq i \leq 6$. As an illustration, using the Tatamibari instance in Fig. 10, we obtain

$$\begin{aligned} C = \big[& [-1,-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1,-2], \\ & [-1,-1,-1,-1,-1,-2,-1],\ldots,[-1,-1,-1,-1,-1,-2,-2],\ldots, \\ & [-1,-1,-2,-2,-2,-2,-2],\ldots,[-2,-2,-2,-2,-2,-1,-2], \\ & [-2,-2,-2,-2,-2,-2,-1],[-2,-2,-2,-2,-2,-2,-2]\big]. \end{aligned} \tag{1}$$

In (1), the first entry of C is obtained by taking -1 from every $H[i]$ for each $0 \leq i \leq 6$, while the last entry of C is obtained by taking -2 from every $H[i]$ for each $0 \leq i \leq 6$. We discuss an algorithm to construct the list C from H in Algorithm 10, which is adapted from [36].

In Algorithm 10 we use a list $indices$ such that $indices[i]$ is used to track the current entry in each $H[i]$ for each $0 \leq i \leq e = mn - h$. This algorithm stores the hints $H[i][indices[i]]$ for all $0 \leq i \leq e = mn - h$ in a temporary list $config$ of size e. Notice that $H[i][indices[i]]$ contains an identity id of the form $\langle symbol \rangle \langle number \rangle$, and thus we assume that the amortized time complexity for adding $H[i][indices[i]]$ to $config$ is $O(1)$. Observe that at the i-th iteration of the for loop in lines 8-11, $config$ is a list that is updated with one possible identity for the i-th empty cell, and thus finally $config = [id_0, id_1, \ldots, id_{e-1}]$ where id_k is a possible identity of the k-th empty cell $(0 \leq k \leq e-1)$. In line 12, the previous list $config$ obtained in lines 8-11 is added as an entry of C, here C contains several lists of possible identities for all empty cells in the board. Notice that since $config$ is of size e, adding $config$ to

C in line 12 takes at most $O(e)$ amortized time. Moreover, when the algorithm terminates, we have $C = [config_0, config_1, \dots, config_d]$ for some $d \leq h^e$, such that $config_i = [id_{i,0}, id_{i,1}, \dots, id_{i,e-1}]$ where $e = mn - h$ and $id_{i,j}$ denotes the possible identity for the j-th empty cell according to the i-th configuration $config_i$ where $0 \leq i \leq d$ and $0 \leq j \leq e - 1$. In other words, C is a list containing at most h^e lists, where each list is of size e and the j-th entry of such a list represents a possible identity for the j-th empty cell. This analysis also infers that the space requirement for C is bounded above by $e \cdot h^e = (mn - h) \cdot h^{mn-h} = O(mn \cdot h^{mn-h})$.

Algorithm 10 Combination(H) creates a list containing lists of possible identities for all empty cells.

Input: A list H as described in the output of Algorithm 9.
Output: A list C containing lists of all possible identities for all empty cells, an entry of C is a list of size $e = mn - h$ and is associated with a unique Tatamibari configuration.

```
1: $p \leftarrow$ length of $H$                              ▷ here $p = e = mn - h$
2: $C \leftarrow$ an empty list
3: for $i \leftarrow 0$ to $p - 1$ do
4:     $indices[i] \leftarrow 0$      ▷ $indices[i]$ helps us to track the current element in each $H[i]$
5: end for
6: while true do
7:     $i \leftarrow 0$
8:     while $i < p$ do
9:         add $H[i][indices[i]]$ to $config$   ▷ adding a possible identity for the $i$-th empty cell to $config$
10:        $i \leftarrow i + 1$
11:    end while
12:    add $config$ to $C$                 ▷ storing possible configurations $config$ to $C$
13:    $next \leftarrow p - 1$
14:    while $next \geq 0$ and $indices[next] + 1 \geq$ length of $H[next]$ do
15:        $next \leftarrow next - 1$
16:    end while
17:    if $next < 0$ then
18:        return $C$
19:    end if
20:    $indices[next] = indices[next] + 1$
21:    $i \leftarrow next + 1$
22:    while $i < p$ do
23:        $indices[i] \leftarrow 0$
24:        $i \leftarrow i + 1$
25:    end while
26: end while
```

The worst-case condition of this Algorithm 10 happens when each $H[i]$ contains exactly h members, which makes C comprises h^e lists of possible configurations. As an example, the list C obtained from the Tatamibari instance in Fig. 10 contains $2^7 = 128$ lists, where each list contains exactly seven members, each member is either -1 or -2. Thus, the asymptotic upper bound for the running time of Algorithm 10 is $O(h^e) = O(h^{mn-h})$. For related analysis regarding Algorithm 10, see, e.g., [37].

Each component of the list C corresponds to a unique Tatamibari configuration. For example, consider the following possible entries of C for the Tatamibari instance given in Fig. 10:

$$\begin{aligned} C_1 &= [-1, -1, -1, -1, -1, -1, -1], \\ C_2 &= [-1, -1, -1, -1, -1, -2, -2], \\ C_3 &= [-1, -1, -2, -2, -2, -2, -2], \\ C_4 &= [-2, -2, -2, -2, -2, -2, -2]. \end{aligned}$$

The lists C_1, C_2, C_3, and C_4 respectively correspond to the Tatamibari configurations in Fig. 11a, Fig. 11b, Fig. 11c, and Fig. 11d. From these configurations, only C_2 and C_3 are Tatamibari solutions.

E. Exhaustive Search Approach for Solving Arbitrary Tatamibari Instances

Our proposed exhaustive search-based Tatamibari solver algorithm is described in Algorithm 11. This algorithm takes an arbitrary Tatamibari instance of size $m \times n$ as an input (represented as a two-

-1	-1	-1
-1	-1	-1
-1	-1	-2

(a) The Tatamibari configuration that corresponds to C_1.

-1	-1	-1
-1	-1	-1
-2	-2	-2

(b) The Tatamibari configuration that corresponds to C_2.

-1	-1	-1
-2	-2	-2
-2	-2	-2

(c) The Tatamibari configuration that corresponds to C_3.

-1	-2	-2
-2	-2	-2
-2	-2	-2

(d) The Tatamibari configuration that corresponds to C_4.

Fig. 11: Tatamibari configurations correspond to C_1, C_2, C_3, and C_4. The gray cells denote the positions of the initial hints.

dimensional array B whose entries are characters of the form $+$, $-$, $|$, or $*$, where $*$ represents an empty cell). The algorithm performs the following steps:

1) appending a number to every hint (non-empty cell) in B using IDFIER function described in Algorithm 5,
2) determining the possible identities (possible regions) of every empty cell in B using POSSIBLE-EMPTY function described in Algorithm 9,
3) generating all possible Tatamibari configurations from the list of all possible identities (possible regions) for all empty cells using COMBINATION function described in Algorithm 10,
4) checking whether each Tatamibari configuration obtained in step 3 is also a Tatamibari solution using VERIFIER function described in Algorithm 4.

The objective of Algorithm 11 is to print every possible Tatamibari solution to an arbitrary Tatamibari configuration B of size $m \times n$.

Algorithm 11 EXHAUSTIVE(B, m, n) prints every possible Tatamibari solution to an arbitrary Tatamibari configuration B represented as a two-dimensional array of size $m \times n$.

Input: A Tatamibari instance B represented as a two dimensional array of size $m \times n$, the entries of the array are either $+$, $-$, $|$, or $*$.
Output: Every two-dimensional array that is a solution to the Tatamibari instance B described in the input (if any solution exists).

```
 1: $B \leftarrow$ IDFIER$(B, m, n)$    ▷ appending a number to every hint using Algorithm 5
 2: $(E, H) \leftarrow$ POSSIBLE-EMPTY$(B)$    ▷ determining the positions of empty cells and the list of possible identities
    for such cells
 3: $C \leftarrow$ COMBINATION$(H)$    ▷ generating all possible Tatamibari configurations using Algorithm 10
 4: $i \leftarrow 0$
 5: while $i <$ length of $C$ do
 6:     $j \leftarrow 0$
 7:     while $j <$ length of $E$ do
 8:         $B[E[j]][0][E[j]][1] \leftarrow C[i][j]$    ▷ filling $B$ with possible identities
 9:         $j \leftarrow j + 1$
10:     end while
11:     $valid \leftarrow$ VERIFIER$(B, m, n)$    ▷ verifying whether the configuration $B$ is also a solution
12:     if $valid$ then
13:         print $B$
14:     end if
15:     $i \leftarrow i + 1$
16: end while
```

The asymptotic upper bound for the running time of Algorithm 11 is discussed in Theorem 1.

Theorem 1. *The asymptotic upper bound for the running time of Algorithm 11 for an arbitrary Tatamibari instance B of size $m \times n$ with h hints where $0 \leq h \leq mn$ is $O(\max\{m^2n^2, h^{mn-h} \cdot hmn\})$.*

Proof. For brevity, we occasionally denote the number of empty cells by e ($e = mn - h$). From the aforementioned analyses, the asymptotic upper bounds for line 1, line 2, and line 3 of Algorithm 11 are respectively $O(mn)$, $O(m^2n^2)$, and $O(h^e)$. Thus, the asymptotic upper bound for lines 1–3 of this algorithm is $O(\max\{m^2n^2, h^{mn-h}\})$. This means if the number of empty cells is sufficiently small (i.e., $h^e < m^2n^2$), then the asymptotic upper bound for lines 1–3 of EXHAUSTIVE(B, m, n) is $O(m^2n^2)$.

Observe that there is a doubly-nested loop in lines 5–16. The number of iterations for the innermost loop in lines 7–10 is equal to the length of E, i.e., the number of empty cells (e). For each possible configuration obtained by replacing the empty cells with particular identities, we check whether such a configuration is also a solution using VERIFIER function described in Algorithm 4 whose upper bound is $O(hmn)$. Assuming that the print procedure in line 13 takes $O(mn)$ time and $e = mn - h < hmn$, the asymptotic upper bound for lines 7–14 is $O(\max\{e, hmn, mn\}) = O(\max\{mn - h, hmn, mn\}) = O(hmn)$. Since the number of possible configurations (the length of the list C) is bounded above by h^e, we conclude that the asymptotic upper bound for lines 5–16 of Algorithm 11 is $O(h^e \cdot hmn) = O(h^{mn-h} \cdot hmn)$.

By combining the asymptotic upper bounds for lines 1–4 and 5–16, we conclude that the asymptotic upper bound for Algorithm 11 is

$$O(\max\{\max\{m^2n^2, h^{mn-h}\}, h^{mn-h} \cdot hmn\}) = O(\max\{m^2n^2, h^{mn-h} \cdot hmn\}). \quad (2)$$

□

Theorem 1 tells us that, if the number of empty cells, $e = mn - h$, is sufficiently small, i.e., $h^{mn-h} < m^2n^2$, then the asymptotic upper bound for the running time of Algorithm 11 is $O(m^2n^2)$. Notice that $O(m^2n^2)$ is the asymptotic upper bound to put a unique number to every hint, check the positions of every empty cell, and determine the list of all possible identities for such a cell. On the other hand, if $e = mn - h$ is sufficiently large, i.e., $m^2n^2 < h^{mn-h}$, then the asymptotic upper bound for the running time of Algorithm 11 is $O(h^{mn-h} \cdot hmn)$. Observe that $O(h^{mn-h} \cdot hmn)$ is the asymptotic upper bound to generate all possible configurations of an $m \times n$ Tatamibari puzzle with h hints and to verify whether each of these configurations is also a Tatamibari solution.

V. Computational Experiments

This section describes the computational experiments of our proposed exhaustive search algorithm for solving arbitrary Tatamibari puzzles. Experiments were performed using C++ programming language and g++ version 11.3.0 compiler on a 64-bit Windows 10 version 21H2 operating system. Here, C++ is chosen because it is relatively faster than other programming languages such as Java or Python [38]. The system also used Intel(R) Core(TM) i7-8550U CPU @ 1.80 GHz with 16.0 GB of RAM. We provide source codes, test cases, and more detailed experimental data used in our experiments for interested readers in `https://github.com/chrisalpha5/TatamibariExhaustive`.

A. Experimental Results for Verification Algorithm

Our proposed verification function in Algorithm 4 was tested against some Tatamibari solutions described in [39]. We considered 26 different Tatamibari solutions where each solution corresponds to a standard English alphabet. Every solution is also represented using a 10×10 array whose entries are of the form $\langle symbol \rangle \langle number \rangle$. Each entry in the array represents an identity of a cell in a Tatamibari configuration. We ran the implementation of our verification algorithm five times for each solution and determined the average running time among these five runs. The average verification time for all Tatamibari solutions is 0.741 milliseconds. The lowest average verification time happened for the solution that corresponds to the letter V (0.566 milliseconds), while the highest average verification time occurred for the solution that corresponds to the letter K (1.092 milliseconds).

B. Counting the Number of Solutions Using Exhaustive Search Techniques

Due to the limitation of our proposed exhaustive search-based algorithm and its implementation in C++, we did not test our algorithm for finding the solutions to Tatamibari instances in [39]. Nevertheless, we tested our proposed algorithm for obtaining the number of possible solutions to empty Tatamibari instances of small sizes. In particular, we removed all hints from the puzzles which are used to get solutions and instead started from an empty grid. We counted the number of ways to fill this grid with valid Tatamibari solutions. Formally, for an empty grid of size $m \times n$, we define $S(m, n)$ as the number of possible valid Tatamibari solutions to such a grid. In addition, we also define $T(m, n)$ as the average running time of three runs for finding the number of valid Tatamibari solutions to such a grid. Our experiments determined the value of $S(m, n)$ and $T(m, n)$ for small m and n, i.e., $m \in \{1, 2\}$ and $n \in \{1, \ldots, 5\}$.

Suppose we consider two distinct Tatamibari instances T_1 and T_2 of the same sizes. These instances may have an identical solution. For example, in Fig. 12 we have two 2×2 Tatamibari instances in Fig. 12a and Fig. 12b with an identical solution given in Fig. 12c.

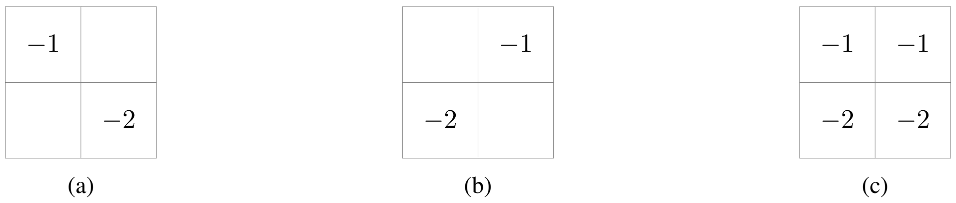


Fig. 12: Two distinct Tatamibari instances with the same solution.

Suppose we consider a list $\mathcal{S}$ containing several distinct solutions to Tatamibari instances of size $m \times n$. Given a particular solution A to a Tatamibari instance of size $m \times n$, we can check whether A exists in $\mathcal{S}$ using the Boolean function Exists described in Algorithm 12. The asymptotic upper bound for the running time of this algorithm is $O(s)$, where s is the number of elements of $\mathcal{S}$.

Algorithm 12 EXISTS($\mathcal{S}, A$) checks if A exists in $\mathcal{S}$.

Input: A list $\mathcal{S}$ containing several distinct solutions to Tatamibari instances of sizes $m \times n$ and a Tatamibari solution A of the same size.
Output: The function returns true if A exists in $\mathcal{S}$, otherwise it returns false.

```
1: $found \leftarrow$ **false**
2: $i \leftarrow 0$
3: **while** $i$ < length of $\mathcal{S}$ **and not** $found$ **do**
4:     **if** $\mathcal{S}[i] = A$ **then**
5:         $found \leftarrow$ **true**
6:     **end if**
7:     $i \leftarrow i + 1$
8: **end while**
9: **return** $found$
```

Suppose we consider two distinct Tatamibari instances T_1 and T_2 whose solutions are respectively S_1 and S_2. Two Tatamibari solutions S_1 and S_2 of the same size are considered identical if the partitions that are associated with S_1 and S_2 are identical regardless of the number put for the hints. In other words, S_1 and S_2 are considered identical if S_2 can be obtained from S_1 by permuting the number associated with every symbol in S_1. For example, in Fig. 13, both solutions in Fig. 13c and Fig. 13d are considered identical because they are associated with the same partition in Fig. 13e.

$+1$		
		$\vert 2$
-3		

(a) A Tatamibari instance of size 3×3.

		$\vert 1$
$+2$		
		-3

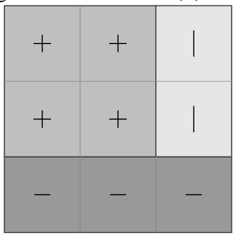
(b) A Tatamibari instance of size 3×3 that is different from Fig. 13a.

$+1$	$+1$	$\vert 2$
$+1$	$+1$	$\vert 2$
-3	-3	-3

(c) A solution to the Tatamibari instance in Fig. 13a.

$+2$	$+2$	$\vert 1$
$+2$	$+2$	$\vert 1$
-3	-3	-3

(d) A solution to the Tatamibari instance in Fig. 13b.



(e) A partition created by the Tatamibari solutions in Fig. 13c and Fig. 13d.

Fig. 13: Example of identical solutions to different instances.

Given a list $\mathcal{S}$ of distinct Tatamibari solutions of size $m \times n$ and a Tatamibari solution A of the same size, we describe the function SIMILAR($\mathcal{S}, A$) in Algorithm 13 to check whether there is a solution $S_i \in \mathcal{S}$ such that S_i and A represent an identical Tatamibari solution regardless of the numbering of the identity. The asymptotic upper bound for the running time of this algorithm is $O(smn)$, where s is the number of elements in $\mathcal{S}$.

Algorithm 13 SIMILAR($\mathcal{S}, A$) checks if a Tatamibari solution A of size $m \times n$ is identical to one of the Tatamibari solutions in $\mathcal{S}$ containing several distinct Tatamibari solutions of size $m \times n$, the identicalness of the solution is considered from the partitions created by the solutions irrespective of the numbering put to the hints.

Input: A list $\mathcal{S}$ containing several distinct solutions to Tatamibari instances of size $m \times n$ and a Tatamibari instance A of the same size.

Output: The function returns true if A is identical to one of the solutions in $\mathcal{S}$ regardless of the numbering put to the hints.

```
1: $i \leftarrow 0$
2: while $i$ < length of $\mathcal{S}$ do
3:     $b \leftarrow$ true
4:     $j \leftarrow 0$
5:     while $j <$ length of $\mathcal{S}[i]$ and $b$ do
6:         if $\mathcal{S}[i][j][0] \neq A[j][0]$ then          ▷ checks if $\mathcal{S}[i]$ and $A$ do not have identical symbols of hints
7:             $b \leftarrow$ false
8:         end if
9:         $j \leftarrow j + 1$
10:    end while
11:    if $b$ then                                              ▷ $\mathcal{S}[i]$ and $A$ have identical symbols of hints
12:        initialize an array $t$ with '0'
13:        $j \leftarrow 0$
14:        while $j <$ length of $\mathcal{S}[i]$ do
15:            if $t[\mathcal{S}[i][j][1]] =$ '0' then
16:                $t[\mathcal{S}[i][j][1]] \leftarrow A[j][1]$         ▷ store the identity number of $\mathcal{S}[i]$ in $t$
17:            end if
18:            $j \leftarrow j + 1$
19:        end while
20:        $j \leftarrow 0$
21:        while $j <$ length of $A$ do
22:            $A[j][1] \leftarrow t[A[j][1]]$                   ▷ assign the corresponding identity of $A$ in $t$
23:            $j \leftarrow j + 1$
24:        end while
25:        if $\mathcal{S}[i] = A$ then
26:            return true
27:        end if
28:    end if
29:    $i \leftarrow i + 1$
30: end while
31: return false
```

Suppose we consider a Tatamibari instance B of size $m \times n$. We describe a modified exhaustive search technique in Algorithm 14 for finding all possible distinct Tatamibari solutions to B and storing all of these solutions in a solution list $\mathcal{SL}$. The functions EXISTS in Algorithm 12 and SIMILAR in Algorithm 13 are used to ensure that all solutions in $\mathcal{SL}$ are distinct and no solutions resemble identical partitions. The asymptotic upper bound for the running time of MODIFIEDEXHAUSTIVE in Algorithm 14 is identical to that of Algorithm 11, namely $O(\max\{m^2n^2, h^{mn-h} \cdot hmn\})$, for finding all distinct solutions to a Tatamibari instance of size $m \times n$ with h hints.

Algorithm 14 ModifiedExhaustive($B, m, n, \mathcal{SL}$) finds every possible Tatamibari solution to an arbitrary Tatamibari configuration B represented as a two-dimensional array of size $m \times n$ and stores the solution in a list $\mathcal{SL}$. The solutions are guaranteed to represent different Tatamibari partitions.

Input: A Tatamibari instance B represented as a two dimensional array of size $m \times n$, the entries of the array are either $+$, $-$, $|$, or $*$, and a list $\mathcal{SL}$ containing the solutions to B.
Output: An updated list $\mathcal{SL}$ containing every two-dimensional array that is a solution to the Tatamibari instance B described in the input (if any solution exists). All solutions are guaranteed to represent different Tatamibari partitions.

```
 1: $B \leftarrow$ Idfier($B, m, n$)                ▷ appending a number to every hint using Algorithm 5
 2: $(E, H) \leftarrow$ Possible-Empty($B$)  ▷ determining the positions of empty cells and the list of possible identities
    for such cells
 3: $C \leftarrow$ Combination($H$)          ▷ generating all possible Tatamibari configurations using Algorithm 10
 4: $i \leftarrow 0$
 5: while $i <$ length of $C$ do
 6:     $j \leftarrow 0$
 7:     while $j <$ length of $E$ do
 8:         $B[E[j]][0][E[j]][1] \leftarrow C[i][j]$          ▷ filling $B$ with possible identities
 9:         $j \leftarrow j + 1$
10:     end while
11:     if Verifier($B, m, n$) then
12:         if $\mathcal{SL}$ is not empty then
13:             if not Exist($\mathcal{SL}, B$) and not Similar($\mathcal{SL}, B$) then
14:                 add $B$ to $\mathcal{SL}$          ▷ adding the solution $B$ to the solution list $\mathcal{SL}$
15:             end if
16:         else
17:             add $B$ to $\mathcal{SL}$              ▷ $B$ is the first solution to the instance
18:         end if
19:     end if
20:     $i \leftarrow i + 1$
21: end while
22: return $\mathcal{SL}$
```

To find the number of different ways for filling an $m \times n$ grid with Tatamibari solutions, we need to generate all possible Tatamibari instances of size $m \times n$. Notice that any Tatamibari instance of size $m \times n$ can be represented using two-dimensional array of size $m \times n$ whose entries are either $+$, $-$, $|$, or $*$ ($*$ signifies an empty cell). If we ignore an empty instance (an instance with no hints), then there are $4^{mn} - 1$ possible ways to create an $m \times n$ Tatamibari instance. We first describe Algorithm 15 which constructs a one-dimensional list L of size mn whose entries are the list $[*, |, -, +]$. The output of this algorithm is then used as an input of Algorithm 10 to construct all $m \times n$ Tatamibari instances with h hints where $1 \leq h \leq mn$. The asymptotic upper bound for the running time of Algorithm 15 is $O(mn)$.

Algorithm 15 Initiate-Empty(m, n) constructs a one-dimensional list L of size mn whose entries are the list $[*, |, -, +]$.

Input: Two integers m and n respectively represent the number of rows and columns in a Tatamibari puzzle.
Output: A one-dimensional list L of size mn whose entries are the list $[*, |, -, +]$.

```
 1: $i \leftarrow 0$
 2: while $i < m$ do
 3:     $j \leftarrow 0$
 4:     while $j < n$ do
 5:         add $[*, |, -, +]$ to $L$
 6:         $j \leftarrow j + 1$
 7:     end while
 8:     $i \leftarrow i + 1$
 9: end while
10: return $L$
```

For example, if $m = n = 2$, then Algorithm 15 produces the list $L = [[*, |, -, +], [*, |, -, +], [*, |, -, +], [*, |, -, +]]$. Notice that, using this list as the input for the function Combination in Algorithm 10 yields a list C containing every list of four entries, each is associated with a possible Tatamibari instance of size 2×2 with 0, 1, 2, 3, or 4 hints, namely

$$\begin{aligned} C = \big[&[*,*,*,*],[*,*,*,|],[*,*,|,|],\ldots,[*,*,+,+],\ldots,\\ &[*,|,|,|],\ldots,[*,+,+,+],\ldots,[+,+,+,-],[+,+,+,+]\big]. \end{aligned} \tag{3}$$

Algorithm 16 describes a method to generate all possible distinct solutions to an empty Tatamibari instance of size $m \times n$. These solutions are stored in a list $\mathcal{SL}$. The algorithm is described as follows. Initially, $\mathcal{SL}$ is defined as an empty list. To construct all possible Tatamibari instances of size $m \times n$, we call the function Initiate-Empty(m, n) and generate all 4^{mn} lists of size mn whose entries are either $*$, $|$, $-$, or $+$ using the function Combination. Notice that $C[0]$ is a one-dimensional list of size mn whose entries are all $*$, which corresponds to an empty Tatamibari instance. Thus, we discard $C[0]$ and only consider the lists $C[1], C[2], \ldots, C[4^{mn-1}], C[4^{mn}]$, i.e., the lists that correspond to Tatamibari instances with at least one hint. At the end of the iteration, $\mathcal{SL}$ contains all possible Tatamibari solutions to an $m \times n$ grid and the size of $\mathcal{SL}$ is the number of ways to construct Tatamibari solutions in an $m \times n$ grid.

Algorithm 16 CountSolutionExhaustive(m, n) counts the number of distinct solutions to an empty $m \times n$ Tatamibari board using exhaustive search technique.

Input: Two integers m and n respectively denoting the number of rows and columns of the Tatamibari board.
Output: The number of ways to fill $m \times n$ Tatamibari board with valid Tatamibari solutions.

```
1: $\mathcal{SL} \leftarrow$ an empty list
2: $L \leftarrow$ Initiate-Empty$(m, n)$
3: $C \leftarrow$ Combination$(L)$
4: $i \leftarrow 1$
5: while $i <$ length of $C$ do                     ▷ $C[0]$ corresponds to an empty instance
6:     change $C[i]$ to a Tatamibari Instance $B_i$
7:     $\mathcal{SL} \leftarrow$ ModifiedExhaustive$(B_i, m, n, \mathcal{SL})$
8: end while
9: print (length of $\mathcal{SL}$)
```

Since the running time for ModifiedExhaustive is bounded by $O(\max\{m^2n^2, h^{mn-h} \cdot hmn\})$ and the length of C is 4^{mn}, then the asymptotic upper bound for Algorithm 16 is $O(\max\{4^{mn}m^2n^2, 4^{mn} \cdot h^{mn-h} \cdot hmn\})$. We test Algorithm 14 to find the number of solutions of several Tatamibari instances of sizes $m \times n$, i.e., $S(m, n)$, for $m \in \{1, 2\}$ and $n \in \{1, \ldots, 5\}$, with at least one hint. Furthermore, since $S(m, n) = S(n, m)$, we only consider the case where $m \leq n$. These values are summarized in Table I. For example, since $S(2, 2) = 7$, we have seven distinct Tatamibari solutions of size 2×2. These solutions are depicted in Fig. 14.

m \ n	1	2	3	4	5
1	1	2	4	8	16
2		7	36	183	898

TABLE I: The value of $S(m, n)$ for several m and n where $m \in \{1, 2\}$ and $n \in \{1, \ldots, 5\}$.

Notice that in Fig. 14, all solutions represent different partitions, but some of them are identical up to rotation (90° clockwise or counterclockwise rotation, 180° clockwise or counterclockwise rotation), reflection (reflection along the horizontal or vertical line), or combination of both rotation and reflection. Given two Tatamibari solutions T_1 and T_2 of the same size, we say T_1 and T_2 are *equivalent* if T_2 can be obtained from T_1 by performing one or more of the following operations and their combinations:

1) 90° clockwise rotation,
2) 90° counterclockwise rotation,
3) 180° clockwise or counterclockwise rotation,
4) horizontal reflection (i.e., flipping the columns),
5) vertical reflection (i.e., flipping the rows).

For instance, in Fig. 14, the solutions in Fig. 14a, Fig. 14b, Fig. 14c, and Fig. 14d are equivalent. Furthermore, by considering these equivalent classes, we have three different Tatamibari solutions in a 2×2 grid, namely the solution with three partitions (Fig. 14a, Fig. 14b, Fig. 14c, and Fig. 14d), the solution with two partitions (Fig. 14e and Fig. 14f), and the solution with one partition (Fig. 14g). Suppose we define $S_{unique}(m, n)$ as the number of different Tatamibari solutions in an $m \times n$ up to the aforementioned rotations and reflections. The value of $S_{unique}(m, n)$ for $m \in \{1, 2\}$ and $n \in \{1, \ldots, 5\}$ is summarized in Table II.

n / m	1	2	3	4	5
1	1	2	3	6	10
2		3	20	78	303

TABLE II: The value of $S_{unique}(m, n)$ for several m and n where $m \in \{1, 2\}$ and $n \in \{1, \ldots, 5\}$.

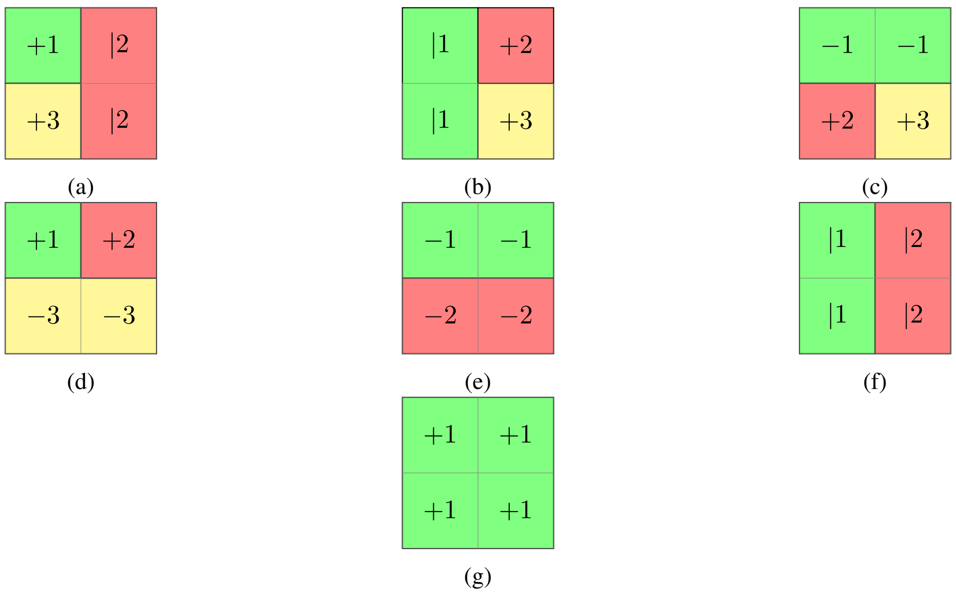


Fig. 14: All possible solutions of the 2×2 empty Tatamibari puzzle.

For a grid of size $m \times n$, we define $T(m, n)$ as the average running time of three runs for finding the number of valid Tatamibari solutions to such a grid. We measured the values of $T(1, n)$ for $n \in \{1, \ldots 10\}$ and $T(2, n)$ for $n \in \{2, \ldots, 5\}$. The average running time for finding the number of solutions to an empty Tatamibari instance of size $1 \times n$ where $1 \leq n \leq 10$ is depicted in Fig. 15, while the average running time for determining the number of solutions to an empty Tatamibari instance of size $2 \times n$ where $2 \leq n \leq 5$ is depicted in Fig. 16. We do not conduct further experiments due to the limitations of our computing environment. However, these empirical results concur with the fact that finding the solutions to Tatamibari instances using the exhaustive search technique requires an exponential amount of time with respect to the size of the grid and the number of hints as described in Theorem 1.

VI. Conclusion and Future Works

We have discussed an exhaustive search approach for solving an arbitrary Tatamibari puzzle. In doing so, we also provide an $O(hmn)$ time algorithm for verifying whether an $m \times n$ Tatamibari configuration with h hints satisfies four rules of the Tatamibari puzzle. In Theorem 1, we prove that the asymptotic upper bound for the running time of our proposed method is $O(\max\{m^2n^2, h^{mn-h} \cdot hmn\})$, where $m \times n$ is the size of the grid and h is the number of the hints in the puzzle. To the best of our knowledge, this result also provides the first explicit upper bound for solving arbitrary Tatamibari puzzles using algorithms that can be implemented without using any specific libraries. Our approach might not be as efficient as

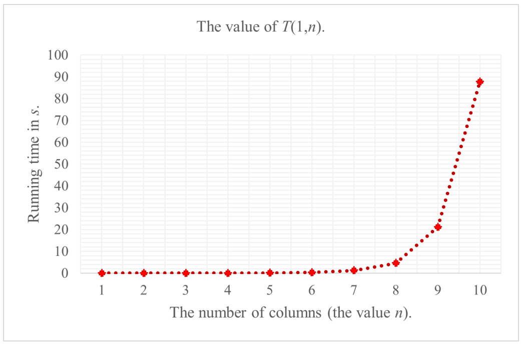


Fig. 15: The average running time of three runs for finding the number of solutions to empty Tatamibari instances of size $1 \times n$ where $1 \leq n \leq 10$.

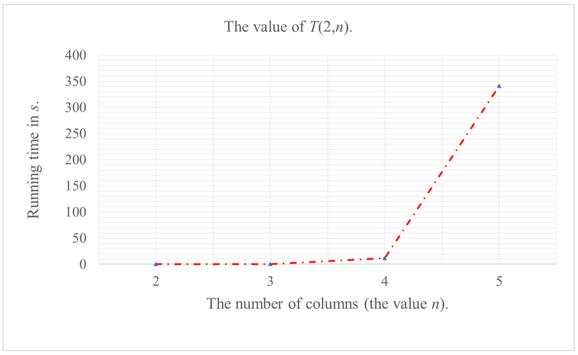


Fig. 16: The average running time of three runs for finding the number of solutions to empty Tatamibari instances of size $2 \times n$ where $2 \leq n \leq 5$.

the method proposed by Adler et al. [25]. However, our exhaustive search technique does not require an additional library so that it can be directly used in any standard imperative programming language. Moreover, we believe that the theoretical and experimental comparisons between the exhaustive search algorithm and SAT-based (including SMT solver) technique need further exploration.

From an experimental point of view, the test cases used in our research are very limited. This happens partly because of the limitation of our computing environment. In addition, we believe that some theoretical adjustment regarding Algorithm 16 is required to determine the value of $S(m, n)$ for larger m and n. For example, one can determine the maximum number of hints in an $m \times n$ Tatamibari puzzle so that such a puzzle has a solution. Another interesting open problem is determining the minimum number of hints so that a Tatamibari instance is guaranteed to have a unique solution.

References

[1] Nikoli, “Puzzle Communication Nikoli: Omopa List,” https://www.nikoli.co.jp/ja/publication/various/nikoli/omopalist/, Sep. 2022, accessed: 2022-09-19.

[2] 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

[3] A. Allen and A. Williams, “Sto-Stone is NP-Complete,” in *CCCG*, 2018, pp. 28–34.

[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] R. A. Hearn and E. D. Demaine, *Games, puzzles, and computation*. CRC Press, 2009.

[7] 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.

[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] A. Ishibashi, Y. Sato, and S. Iwata, "NP-completeness of two pencil puzzles: Yajilin and Country Road," *Utilitas Mathematica*, vol. 88, pp. 237–246, 2012.

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

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

[12] C. Iwamoto and T. Ide, "Five Cells and Tilepaint are NP-Complete," *IEICE TRANSACTIONS on Information and Systems*, vol. 105, no. 3, pp. 508–516, 2022.

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

[14] C. Iwamoto, M. Haruishi, and T. Ibusuki, "Herugolf and Makaro are NP-complete," in *9th International Conference on Fun with Algorithms (FUN 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[15] 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.

[16] D. Andersson, "Hiroimono is NP-complete," in *International Conference on Fun with Algorithms*. Springer, 2007, pp. 30–39.

[17] 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.

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

[19] C. Iwamoto and T. Ide, "Moon-or-Sun, Nagareru, and Nurimeizu are NP-complete," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, p. 2021DMP0006, 2022.

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

[21] 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.

[22] C. Iwamoto and M. Haruishi, "Computational complexity of Usowan puzzles," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 101, no. 9, pp. 1537–1540, 2018.

[23] 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.

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

[25] A. Adler, J. Bosboom, E. D. Demaine, M. L. Demaine, Q. C. Liu, and J. Lynch, "Z3-based Tatamibari solver, and figures from Tatamibari NP-hardness paper," https://github.com/jbosboom/tatamibari-solver, Feb. 2020, accessed: 2022-07-28.

[26] J. J. W. Bosboom, "Exhaustive search and hardness proofs for games," Ph.D. dissertation, Massachusetts Institute of Technology, 2020.

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

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

[29] U. Pfeiffer, T. Karnagel, and G. Scheffler, "A Sudoku-Solver for Large Puzzles using SAT," in *LPAR short papers (Yogyakarta)*, 2010, pp. 52–57.

[30] M. Z. Musa, "Interactive Sudoku Solver Using Propositional Logic in Python," Bachelor Thesis, Undergraduate Program of Informatics, School of Computing, Telkom University, 2018.

[31] A. Shaleh, "Solving Shikaku Using Propositional Logic Approach," Bachelor Thesis, Undergraduate Program of Informatics, School of Computing, Telkom University, 2019.

[32] M. d. Berg and A. Khosravi, "Optimal binary space partitions in the plane," in *International Computing and Combinatorics Conference*. Springer, 2010, pp. 216–225.

[33] StackOverflow, "What is Constant Amortized Time?" https://stackoverflow.com/questions/200384/what-is-constant-amortized-time, Oct. 2008, accessed: 2022-12-20.

[34] ——, "Why is the time complexity of Python's list.append() method $O(1)$?" https://stackoverflow.com/questions/33044883/why-is-the-time-complexity-of-pythons-list-append-method-o1, Oct. 2015, accessed: 2022-12-20.

[35] Quora, "What is algorithmic complexity of push_back in std::vector? (Assume default allocator)," https://www.quora.com/What-is-algorithmic-complexity-of-push_back-in-std-vector-Assume-default-allocator, Dec. 2022, accessed: 2022-12-20.

[36] A. Sharma, "Combinations from n arrays picking one element from each array," https://www.geeksforgeeks.org/combinations-from-n-arrays-picking-one-element-from-each-array/, Apr. 2022, accessed: 2022-04-27.

[37] P. Vaillancourt, "Algorithm to generate combinations of n elements from n sets of m elements," https://cs.stackexchange.com/questions/125752/algorithm-to-generate-combinations-of-n-elements-from-n-sets-of-m-elements, May 2020, accessed: 2022-12-23.

[38] L. Prechelt, "Are scripting languages any good? A validation of Perl, Python, Rexx, and Tcl against C, C++, and Java." *Adv. Comput.*, vol. 57, pp. 205–270, 2003.

[39] E. D. Demaine, "Tatamibari Font," https://github.com/edemaine/font-tatamibari, Apr. 2022, accessed: 2022-05-11.