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

Ind. Journal on Computing
Vol. 8, Issue. 2, August 2023. pp. 36-64
doi:10.34818/indojc.2023.8.2.750

Elementary Search-based Algorithms for Solving Tilepaint Puzzles

Vincentius Arnold Fridolin #1, Muhammad Arzaki *2, Gia Septiana Wulandari *2

# *Undergraduate Student, Computing Laboratory*
School of Computing, Telkom University, Indonesia (40257)

1 vincentiusarnoldfridolin@gmail.com

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

2 {arzaki,gia}@telkomuniversity.ac.id

Abstract

We discuss the elementary computational aspects of Tilepaint puzzles, single-player logic puzzles introduced in 1995 and confirmed NP-complete in 2022. We propose two elementary search-based algorithms for solving such puzzles: the complete search technique with a bitmasking approach and the prune-and-search technique with a backtracking approach and pruning optimization. We show that the asymptotic running times of these algorithms for solving an $m \times n$ Tilepaint instance containing p tiles are respectively $O(2^p \cdot p \cdot mn)$ and $O(2^p \cdot mn)$, implying that the latter method is asymptotically faster by a factor of p. We also discuss tractable and intractable variants of the puzzles. We show that an $m \times n$ Tilepaint instance containing mn tiles of size 1×1 is solvable in polynomial time. In contrast, we show that solving general $m \times 1$ and $1 \times n$ Tilepaint puzzles remains intractable by reducing such problems from the subset-sum problem.

Keywords: complete search, complexity, prune-and-search, Tilepaint puzzle, tractable subproblems, reduction

Abstrak

Kami membahas aspek komputasi elementer dari teka teki Tilepaint, teka teki logika pemain tunggal yang diperkenalkan pada tahun 1995 dan dikonfirmasi NP-*complete* pada tahun 2022. Kami mengusulkan dua algoritma elementer berbasis pencarian untuk memecahkan teka teki tersebut: teknik pencarian menyeluruh dengan pendekatan *bitmasking* dan teknik pencarian-dan-pangkas dengan pendekatan runut balik dan optimasi pemangkasan. Kami menunjukkan bahwa waktu eksekusi asimtotik dari algoritma-algoritma ini untuk menyelesaikan teka teki Tilepaint $m \times n$ yang berisi p kelompok ubin (*tiles*) masing-masing adalah $O(2^p \cdot p \cdot mn)$ dan $O(2^p \cdot mn)$, menyiratkan bahwa metode kedua lebih cepat secara asimtotik dengan faktor p. Kami juga membahas varian teka teki tersebut yang bisa diselesaikan cepat (*tractable*) dan tidak (*intractable*). Kami menunjukkan bahwa bahwa teka teki Tilepaint $m \times n$ yang berisi mn ubin berukuran 1×1 dapat diselesaikan dalam waktu polinomial. Sebaliknya, kami menunjukkan bahwa memecahkan teka teki Tilepaint yang berukuran $m \times 1$ dan $1 \times n$ secara umum tetap *intractable* dengan mereduksi masalah tersebut dari masalah jumlahan subhimpunan (*subset-sum problem*).

Kata Kunci: kompleksitas, pencarian-dan-pangkas, pencarian menyeluruh, reduksi, submasalah *tractable*, teka teki Tilepaint

Received on April, 2023. Accepted on August, 2023

I. INTRODUCTION

Tilepaint (タイルペイント) is a logic pencil-and-paper-based puzzle invented by Toshihari Yamamoto in Japan and popularized by Nikoli, a publisher that specializes in logic puzzles.[1] According to Jimmy Goto—then a Nikoli manager—Tilepaint first appeared in issue 53 of Nikoli's quarterly Puzzle Communication magazine in 1995 and has been published regularly ever since [1]. This puzzle considers an $m \times n$ grid of cells where the cells are divided into some *tiles*. A tile (sometimes referred to as a *region*) is a collection of orthogonally connected cells separated by thick lines. Initially, all cells are left blank (uncolored). The player plays the game by ensuring that each cell in every tile is either colored or uncolored (i.e., we must color all cells in a tile or leave all of them uncolored). There are constraints indicated by numbers at the top and left of the grid, specifying the number of cells that must be colored in the corresponding row and column. The problem is to find any configuration that matches the number of colored cells according to the constraint described for each row and column (if any).

Solving puzzles is valuable for enhancing computational thinking, mathematics, and problem-solving skills. Regularly solving puzzles stimulates various cognitive functions, including creativity, critical thinking, and mathematical thinking, which are fundamental to mathematical proficiency [2]. Some single-player puzzles have notable connections to important computational and mathematical problems, sparking the scientific community's interest in exploring these puzzles [3]–[5]. Several one-player puzzles have been confirmed NP-complete, such as (in alphabetical order, the year in which the puzzle is confirmed NP-complete indicated inside the brackets): Five Cells (2022) [6], Juosan (2021) [7], Kurotto (2021) [7], Minesweeper (2000) [8], Moon-or-Sun (2022) [9], Nagareru (2022) [9], Nonogram (1996) [10], Nurimeizu (2022) [9], Path Puzzles (2020) [11], Sudoku (2003) [12], Suguru (2022) [13], Tatamibari (2020) [14], and Yin-Yang (2021) [15].

Tilepaint puzzles were recently confirmed NP-complete in 2022 by Iwamoto and Ide [6]. The NP-completeness of the puzzles means a polynomial time algorithm exists to verify whether a configuration satisfies the puzzle's rules. Moreover, this implies solving a general instance of Tilepaint puzzles currently requires an exponential time algorithm unless P = NP. Nevertheless, as far as we know, there has never been any discussion of further research into the algorithms used to solve these puzzles. Various methods are proposed to solve NP-complete puzzles, including non-elementary techniques such as integer programming model [16] and SAT solver [17]–[19]. This paper discusses two elementary search-based techniques for solving Tilepaint puzzles: a complete search approach with a bitmasking technique and a prune-and-search approach with a backtracking method. Previous studies show that elementary search-based methods can be applied for solving NP-complete puzzles, such as the prune-and-search method for solving Yin-Yang puzzles [20] and the exhaustive search technique for solving Tatamibari puzzles [21].

Tilepaint puzzles are closely related to two-dimensional discrete tomography problems, which relate to constructing binary images from a small number of their projections. The application of discrete tomography problems have been extensively discussed in image processing [22], [23]. Tilepaint puzzles can be extended from these problems by imposing the additional *tile rule*, namely, all cells within the same tile must have the same color. Other NP-complete puzzles related to two-dimensional discrete tomography problems are Nonogram [10] and Path Puzzles [11]. Nevertheless, it is well-known that the two-dimensional discrete tomography problem is solvable in polynomial time if the numerical constraints for all rows and columns in the instance are known [24], [25]. Modifying existing rules and introducing additional rules to the formerly tractable two-dimensional discrete tomography problem transforms this problem into interesting non-trivial NP-complete puzzles. In the case of the Nonogram puzzle, the NP-hardness arises due to multiple numerical constraints for each row and column, while for the Path Puzzles, the NP-hardness occurs due to the Hamiltonicity constraint.

This paper also discusses some tractable and intractable variants of the original Tilepaint puzzles. Studying originally NP-complete problems' variations that are tractable is important in theoretical computer science [26], [27]. This paper shows that an $m \times n$ Tilepaint instance containing mn

[1]An example of a famous puzzle published by Nikoli is Sudoku.

tiles of size 1×1 is solvable in polynomial time if all the numerical constraints are known. In contrast, this paper also shows that reducing the dimension of a Tilepaint puzzle to $m \times 1$ or $1 \times n$ does not necessarily make the puzzle becomes tractable.

We organize the rest of our investigation into the following sections. In Section II, we discuss the formal definition, data structure representation, and mathematical properties of Tilepaint puzzles. We outline the NP-completeness of Tilepaint puzzles from the previous paper [6] and briefly explore a similar problem to the Tilepaint puzzle, i.e., the two-dimensional discrete tomography problems. We explore some important mathematical observations of the Tilepaint puzzle; specifically, we discuss a condition where a Tilepaint puzzle does not have a solution. This condition can be used as a prune condition to optimize the Tilepaint solver. Section III discusses a polynomial time approach to verify whether a Tilepaint configuration satisfies the puzzle's rules. We show that this verification algorithm takes $O(mn)$ time for an $m \times n$ Tilepaint configuration with any number of tiles. Section IV discusses a complete search approach with a bitmasking technique to solve arbitrary Tilepaint puzzles of size $m \times n$ containing p tiles in $O(2^p \cdot p \cdot mn)$ time. The prune-and-search approach with a backtracking technique and pruning optimization for solving arbitrary Tilepaint puzzles of size $m \times n$ with p tiles in $O(2^p \cdot mn)$ time is discussed in Section V. We discuss a tractable variant of Tilepaint puzzles, namely an $m \times n$ puzzle with mn tiles of size 1×1 in Section VI. Moreover, we also show that a general Tilepaint puzzle of size $m \times 1$ or $1 \times n$ remains intractable in Section VII. Section VIII presents the experimental results showcasing the performance of both solver algorithms in solving Tilepaint puzzles of various sizes. Finally, we summarize and conclude our investigation in Section IX.

II. Preliminaries

We use one-based indexing for all arrays throughout this paper. For a one-dimensional array A of size/length n, the notation $A[i]$ denotes the i-th entry of A ($1 \leq i \leq n$) and for a two-dimensional array B of size $m \times n$ (i.e., containing m rows and n columns), the notation $B[i][j]$ denotes the entry in row i and column j of B where $1 \leq i \leq m$ and $1 \leq j \leq n$.

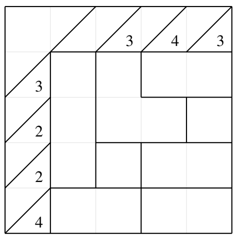
A. Formal Definition and Representation of Tilepaint Puzzles

A Tilepaint instance informally is an empty $m \times n$ grid grouped into several *tiles/regions* (orthogonally connected cells). The configuration is a collection of tiles that are colored (or *blackened*) or uncolored (or *left white*). However, not all configurations are solutions to an instance—a solution to an instance is the configuration that satisfies the puzzle's rules. The following definition formalizes the description of Tilepaint instances, configurations, and solutions.

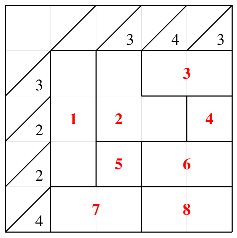
Definition 1. *An instance of a Tilepaint puzzle (or Tilepaint instance) of size $m \times n$ with p tiles/regions is a rectangular grid of m rows and n columns satisfying the following conditions:*

1) *a cell (i, j) is an intersection of row i and column j where $1 \leq i \leq m$ and $1 \leq j \leq n$;*
2) *there are p tiles $T_1, T_2, \ldots, T_p$ where $1 \leq p \leq mn$, here a tile T_i is a collection of orthogonally connected cells;*
3) *all cells are initially uncolored;*
4) *there can be a constraint number for each row i, denoted by CR_i (a non-negative integer between 0 and n, inclusive), signifying the number of colored cells in row i, where $1 \leq i \leq m$;*
5) *there can be a constraint number for each column j, denoted by CC_j (a non-negative integer between 0 and m, inclusive), signifying the number of colored cells in a column j, where $1 \leq j \leq n$.*

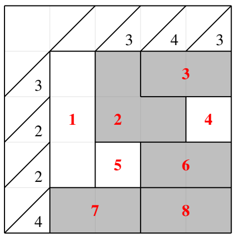
A configuration of a Tilepaint instance is obtained by coloring zero or more cells in the instance. A solution to a Tilepaint instance is a configuration that satisfies the rules of the Tilepaint puzzle, namely, all cells in the same tile must be colored or uncolored, and the number of colored cells in each row and column must comply with the predefined constraint number.



(a) A Tilepaint instance (initial configuration).



(b) An instance in Fig. 1a with numbered tiles.



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

```
(1, 0) (2, 0) (3, 0) (3, 0)
(1, 0) (2, 0) (2, 0) (4, 0)
(1, 0) (5, 0) (6, 0) (6, 0)
(7, 0) (7, 0) (8, 0) (8, 0)
```

(d) An array representation of the Tilepaint instance in Fig. 1b.

```
(1, 0) (2, 1) (3, 1) (3, 1)
(1, 0) (2, 1) (2, 1) (4, 0)
(1, 0) (5, 0) (6, 1) (6, 1)
(7, 1) (7, 1) (8, 1) (8, 1)
```

(e) An array representation of Tilepaint solution in Fig. 1c.

Fig. 1: Tilepaint instance and solution as well as their array representations.

An example of a Tilepaint instance and its corresponding solution are given in Fig. 1. To study the algorithmic methods for solving the Tilepaint puzzle, we use two-dimensional arrays of size $m \times n$ to represent the Tilepaint instance, configuration, and solution. The (i, j) entry of a Tilepaint instance is a pair $(T_{i,j}, 0)$ where $T_{i,j}$ is an integer denoting the tile number to which the cell (i, j) belongs. The tiles are numbered using the row-major order convention, i.e., if there are p tiles, the top-leftmost tile is numbered 1 while the last encountered tile is numbered p. The number 0 in the pair $(T_{i,j}, 0)$ signifies that the cell (i, j) is initially uncolored. We denote a Tilepaint instance represented by this two-dimensional array with IG. A Tilepaint instance with numbered tiles and its two-dimensional array representation are illustrated in Fig 1b and Fig. 1d.

A Tilepaint configuration is represented by a two-dimensional array CG where $CG[i][j] = (T_{i,j}, C_{i,j})$. The definition of $T_{i,j}$ is identical to the previous one for a Tilepaint instance, while $C_{i,j}$ denotes the color status of such a cell ($C_{i,j} = 1$ if and only if the cell (i, j) is colored). A Tilepaint configuration CG is also a Tilepaint solution if it satisfies the aforementioned puzzle's rules.

The number on the top and left sides of the grid are correspondingly represented using two arrays CC of size n and CR of size m. We put column constraint $CC_1, CC_2, \ldots, CC_n$ in array CC and row constraint $CR_1, CR_2, \ldots, CR_m$ in array CR. Recall that the puzzle may not have complete information, i.e., no number is defined on a particular row or column. In this case, the value CR_i or CC_j is defined as -1 for algorithmic purpose. For example, for the Tilepaint instance in Fig. 1a, we have $CC = [-1, 3, 4, 3]$ and $CR = [3, 2, 2, 4]$. The output format of our

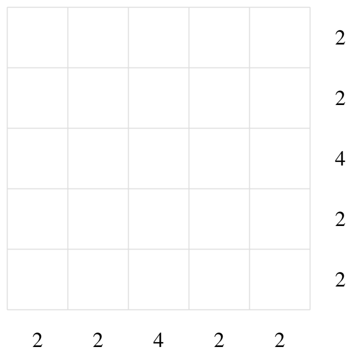
Tilepaint puzzle solver is either a subset of $\{1, 2, \ldots, p\}$ or a notification that the Tilepaint puzzle has no solution. If the puzzle has a solution, the subset of $\{1, 2, \ldots, p\}$ signifies the tile number that must be colored to satisfy the rules. For example, the Tilepaint solution in Fig. 1c is a subset of colored tiles, that is, $\{2, 3, 6, 7, 8\}$.

B. Overview of the NP-completeness of Tilepaint Puzzles

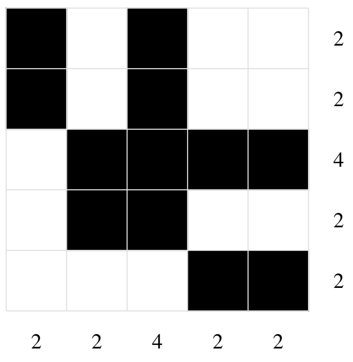
Tilepaint puzzles were recently proven NP-complete by Iwamoto and Ide in 2022 [6]. The puzzles are NP-complete even when the constraint numbers are all defined for all rows and columns (i.e., the puzzles have *complete information* or *complete numerical constraint*). The NP-completeness of Tilepaint puzzles means that checking whether a configuration of an arbitrary Tilepaint puzzle is a solution to the instance can be done in polynomial time in terms of the grid dimension. However, unless P = NP, no known polynomial time algorithm can solve arbitrary Tilepaint instances.

Iwamoto and Ide stated that Tilepaint puzzles can be reduced from the Three-Dimensional Matching Problem (3DM), i.e., a problem for finding the largest three-dimensional matching in a given hypergraph [6]. This problem itself is one of the first problems proven NP-complete by reduction from 3-SAT [28].

Another computational problem that is similar to the Tilepaint puzzle is two-dimensional orthogonal discrete tomography, i.e., a problem of constructing a binary image (represented by black and white pixels or using a binary matrix) from the given number of black pixels (or numbers of ones) in every row and column in the instance. An illustration of a two-dimensional orthogonal discrete tomography problem is given in Fig. 2.



(a) An instance of two-dimensional orthogonal discrete tomography.



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

Fig. 2: An example of two-dimensional orthogonal discrete tomography problem.

According to previously known results in [24], [25], every instance of a two-dimensional orthogonal discrete tomography with complete information—i.e., every row and column has a number constraint—is solvable in polynomial time. We can verify the existence of a solution to such an instance using *majorization* technique and construct such a solution using a greedy algorithm. Notice that a Tilepaint puzzle can be considered as a two-dimensional orthogonal discrete tomography problem with one additional rule: all cells that belong to the same tile must be either colored or uncolored. Surprisingly, this additional rule is the aspect that makes the puzzle NP-hard.

C. Some Important Observations

We first investigate a property regarding Tilepaint puzzles that have complete information. Here, we prove that if the sums of the numerical constraints for the rows and columns of an instance are different, then the instance has no solution. The following theorem is due to Ryser [24]. However, the proof was not given in the original paper.

Theorem 1. *Suppose we consider a Tilepaint puzzle with complete information and let CR_i and CC_j be the numerical constraint for row i and column j, respectively. We define $SR = \sum_{i=1}^{m} CR_i$ and $SC = \sum_{i=j}^{n} CC_j$. The puzzle has no solution if $SR \neq SC$.*

Proof. Based on Definition 1, a cell is an intersection of row i and column j. Therefore, when a cell (i, j) is colored, it is colored in a cell in row i and a cell in column j. Consequently, if the total number of colored cells across all rows is SR, then SR must be equal to SC. If, for instance, $SR > SC$, it would indicate that the colored cell in a row is not part of any column. This is impossible as it would violate the constraints of the puzzle. The same applies in a scenario where $SR < SC$. $\square$

In a Tilepaint puzzle, it is possible to determine whether a tile should be colored or not by checking the parity of the constraint numbers located on the top and the left of the grid. We first consider the following definition regarding the *tile segment* and its corresponding size in a Tilepaint instance.

Definition 2. *Suppose we consider row i ($1 \leq i \leq m$) in a Tilepaint instance of size $m \times n$. Let $H_i \geq 1$ denotes the number of regions that appears in row i. A tile segment of row i is defined as the collection of cells in row i belonging to the same region. We define $TR_i = \{R_{1,i}, R_{2,i}, \ldots, R_{H_i,i}\}$ as the collection of tiles segments in row i. We also define $s_{R_{j,i}}$ as the cardinality of $R_{j,i}$ ($1 \leq j \leq H_i$).*

Analogously, suppose we consider column j ($1 \leq j \leq n$) in a Tilepaint instance of size $m \times n$. Let $V_j \geq 1$ denotes the number of regions that appears in column j. A tile segment of column j is defined as the collection of cells in column j belonging to the same region. We define $TC_j = \{C_{1,j}, C_{2,j}, \ldots, C_{V_j,j}\}$ as the collection of tiles segments in column j. We also define $s_{C_{i,j}}$ as the cardinality of $C_{i,j}$ ($1 \leq i \leq V_j$).

We provide an illustration of Definition 2 using a Tilepaint instance in Fig. 3. Notice that $R_{j,i}$ is the j-th tile segment appearing in row i. In the first row of the Tilepaint instance in Fig. 3, we have $H_1 = 3$ and $TR_1 = \{R_{1,1}, R_{2,1}, R_{3,1}\}$ where $R_{1,1} = \{(1,1), (1,3)\}$, $R_{2,1} = \{(1,2)\}$, and $R_{3,1} = \{(1,4), (1,5)\}$. Therefore, we have $s_{R_{1,1}} = |R_{1,1}| = 2$, $s_{R_{2,1}} = |R_{2,1}| = 1$, and $s_{R_{3,1}} = |R_{3,1}| = 2$. Meanwhile, in the first column of the instance, we have $V_1 = 3$ and $TC_1 = \{C_{1,1}, C_{2,1}, C_{3,1}\}$ where $C_{1,1} = \{(1,1), (2,1)\}$, $C_{2,1} = \{(3,1), (4,1)\}$, and $C_{3,1} = \{(5,1)\}$. Hence, $s_{C_{1,1}} = |C_{1,1}| = 2$, $s_{C_{2,1}} = |C_{2,1}| = 2$, and $s_{C_{3,1}} = |C_{3,1}| = 1$.

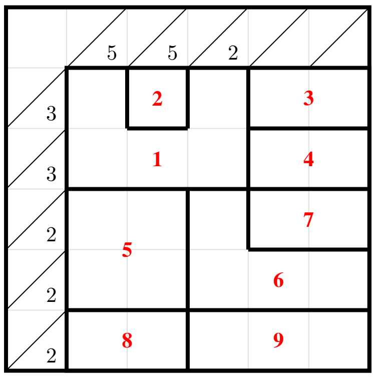


Fig. 3: A Tilepaint instance with 25 cells and nine tiles.

The following definition formalizes the number of colored tile segments in particular rows and columns.

Definition 3. *Suppose we consider a row i and column j where $1 \leq i \leq m$ and $1 \leq j \leq n$. Let mo_i be the number of odd-sized colored tile segments in row i and no_j be the number of odd-sized colored tile segments in column j. Similarly, let me_i be the number of even-sized colored tile segments in row i and ne_j be the number of even-sized colored tile segments in column j.*

The following definition formalizes the collections of colored tile segments of odd and even size in particular rows and columns.

Definition 4. *Let $\{RO_{1,i}, RO_{2,i}, \ldots, RO_{mo_i,i}\}$ and $\{RE_{1,i}, RE_{2,i}, \ldots, RE_{me_i,i}\}$ correspondingly denote the collection of colored tile segments with odd and even size appearing in row i. For each $RO_{\alpha,i}$, we define $so_{\alpha,i}$ as the number of cells in $RO_{\alpha,i}$ where $1 \leq \alpha \leq mo_i$. We also define $se_{\beta,i}$ as the number of cells within $RE_{\beta,i}$ where $1 \leq \beta \leq me_i$. Similarly, let $\{CO_{1,j}, CO_{2,j}, \ldots, CO_{no_j,j}\}$ and $\{CE_{1,j}, CE_{2,j}, \ldots, CE_{ne_j,j}\}$ correspondingly denote the collection of colored tile segments with odd and even size appearing in column j. For each $CO_{\gamma,j}$, we define $so_{\gamma,j}$ as the number of cells in $CO_{\gamma,j}$ where $1 \leq \gamma \leq no_j$. We also define $se_{\delta,j}$ as the number of cells within $CE_{\delta,j}$ where $1 \leq \delta \leq ne_j$.*

We illustrate the notations in Definition 3 and Definition 4 in the following example.

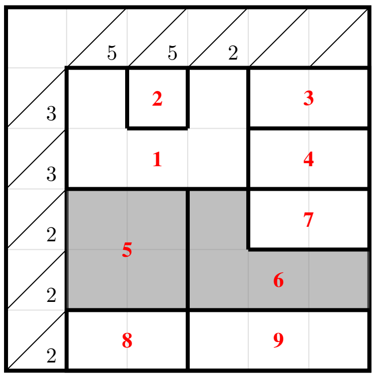


Fig. 4: A Tilepaint configuration of the instance in Fig. 3

Example 1. *Suppose we consider a configuration in Fig. 4 and observe the third row. We have $H_3 = 3$ and there are three tile segments in this row, namely, $R_{1,3} = \{(3,1),(3,2)\}$, $R_{2,3} = \{(3,3)\}$, and $R_{3,3} = \{(3,4),(3,5)\}$. Thus, $mo_3 = 1$ because we only have one odd-sized colored tile segment in this row, i.e., $R_{2,3}$. We also have $me_3 = 1$ because we only have one even-sized colored tile segment in this row, i.e., $R_{1,3}$. Moreover, we have $RO_{1,3} = R_{2,3} = \{(3,3)\}$ and $RE_{1,3} = R_{1,3} = \{(3,1),(3,2)\}$. Consequently, $so_{1,3} = 1$ and $se_{1,3} = 2$.*

If we consider the third column, we have $V_3 = 3$ and there are three tile segments in this column, namely, $C_{3,1} = \{(1,3),(2,3)\}$, $C_{3,2} = \{(3,3),(4,3)\}$, and $C_{3,3} = \{(5,3)\}$. We have $no_3 = 0$ because there is no odd-sized colored tile segment in this column. However, we have $ne_3 = 1$ because there is one even-sized colored tile segment in this column, i.e., $C_{3,2}$. We have $CE_{3,2} = C_{3,2} = \{(3,3),(4,3)\}$ and $se_{3,2} = 2$.

Here, we prove that if CR_i is odd, then there is an odd number of $RO_{\alpha,i}$ that is colored (i.e., there is an odd number of odd-sized tile segments that is colored in row i).

Theorem 2. *Suppose we consider a row i where $1 \leq i \leq m$ and there are H_i tile segments $R_{1,i}, R_{2,i}, \ldots, R_{H_i,i}$ in this row. Suppose CR_i is the constraint number for row i. If CR_i is odd, then there is an odd number of colored tile segments in row i where each collection of these tile segments is of an odd size.*

Proof. We prove the theorem by considering its contrapositive: if there is an even number of colored tile segments where each tile segment is of an odd size, then CR_i must be even. Notice that $so_{\alpha,i}$ is odd and $se_{\beta,i}$ is even for every $1 \leq \alpha \leq mo_i$ and $1 \leq \beta \leq me_i$. Let us assume that we have an even number of colored tile segments of odd size in row i, that is, mo_i is even using the notation in Definition 3. We can write $so_{\alpha,i} = 2k_\alpha + 1$ and $se_{\beta,i} = 2\ell_\beta$ where $k_\alpha, \ell_\beta \in \mathbb{Z}$. Observe that the constraint CR_i equals the number of colored cells in row i, i.e., the number of cells in the colored tile segments of odd and even sizes, thus

$$\begin{aligned} CR_i &= \left(\sum_{\alpha=1}^{mo_i} so_{\alpha,i}\right) + \left(\sum_{\beta=1}^{me_i} se_{\beta,i}\right) \\ &= \left(\sum_{\alpha=1}^{mo_i} 2k_\alpha + 1\right) + \left(\sum_{\beta=1}^{me_i} 2\ell_\beta\right) \\ &= 2(k_1 + k_2 + \cdots + k_{mo_i}) + mo_i + 2(\ell_1 + \ell_2 + \cdots + \ell_{me_i}) \end{aligned}$$

Since we assume mo_i is even, $mo_i = 2p$ for some $p \in \mathbb{Z}$. Hence, we have,

$$\begin{aligned} CR_i &= 2(k_1 + k_2 + \cdots + k_{mo_i}) + 2p + 2(\ell_1 + \ell_2 + \cdots + \ell_{me_i}) \\ &= 2(k_1 + k_2 + \cdots + k_{mo_i} + p + \ell_1 + \ell_2 + \cdots + \ell_{me_i}). \end{aligned}$$

Therefore CR_i is even if mo_i is even. □

This theorem is useful for backtracking to determine which tile segments should be colored in a row. Suppose the coloring process is performed from left to right. Here, if the number of cells that have been colored is even and CR_i is odd, then there must be an odd number of odd-sized tile segments that must be colored to meet the parity of CR_i. The following corollary is an immediate analogy of Theorem 2.

Corollary 1. *Suppose we consider a column j where $1 \leq j \leq n$ and there are V_j tile segments $C_{j,1}, C_{j,2}, \ldots, C_{V_j,j}$ in this column. Suppose CC_j is the constraint number for column j. If CC_j is odd, then there is an odd number of colored tile segments in column j where each collection of these tile segments is of an odd size.*

In the following observation, we prove that if a constraint in a specific row or column is positive but less than the sizes of all tile segments occurring in that row or column, then the corresponding Tilepaint instance has no solution.

Theorem 3. *Suppose there are H_i tile segments in row i, namely, $\{R_{i,1}, R_{i,2}, \ldots, R_{i,H_i}\}$. Let $s_{i,j}$ denotes the size of tile segment j in row i for $1 \leq j \leq H_i$. If CR_i satisfies $0 < CR_i < s_{i,j}$ for all $1 \leq j \leq H_i$, then the corresponding Tilepaint instance has no solution.*

Proof. We prove the theorem by contradiction: suppose $0 < CR_i < s_{i,j}$ for all $1 \leq j \leq H_i$, but the Tilepaint instance has a solution. Suppose $A = \{1, 2, \ldots, H_i\}$ denotes the set representing the label of the tile segments. Suppose we choose $0 < m \leq H_i$ tile segments to be colored, and the label of these chosen tile segments are represented by $B = \{a_1, a_2, \ldots, a_m\}$. Clearly $\emptyset \subset B \subseteq A$. Then we have $CR_i = s_{i,a_1} + s_{i,a_2} + \cdots + s_{i,a_m}$. Since we assume $CR_i < s_{i,j}$ for all $1 \leq j \leq H_i$, we also have $CR_i < s_{i,a_1} + s_{i,a_2} + \cdots + s_{i,a_m}$, which is a contradiction. □

The following corollary is an immediate analogy of Theorem 3.

Corollary 2. *Suppose there are V_j tile segments in column j, namely $C_{i,j}, C_{2,j}, \ldots, C_{V_j,j}$. Let $s_{k,j}$ denotes the size of tile segment k in column j for $1 \leq k \leq V_j$. If CC_j satisfies $0 < CC_j < s_{k,j}$ for all $1 \leq k \leq V_j$, then the corresponding Tilepaint instance has no solution.*

III. POLYNOMIAL TIME ALGORITHM FOR VERIFYING TILEPAINT SOLUTIONS

Verifying whether a Tilepaint configuration is a valid solution can be performed in polynomial time. To verify a Tilepaint configuration, we must check if the configuration satisfies the following rules:

1) If the puzzle has complete information, then $\sum_{i=1}^{m} CR_i$ must be equal to $\sum_{j=1}^{n} CC_j$ according to Theorem 1.

2) All cells in the same tile must be colored or not colored.
3) The number of colored cells in row i must be equal to CR_i and the number of colored cells in column j must be equal to CC_j for $1 \leq i \leq m$ and $1 \leq j \leq n$.

A. Checking Constraint Related to the Sum of Numbers on Row and Column

Suppose we consider a Tilepaint puzzle with complete information. To check the compliance of the first rule, that is, $\sum_{i=1}^{m} CR_i = \sum_{j=1}^{n} CC_j$, we first need to ensure that the numerical constraints for all rows and columns are defined. To check it, we can iterate through two arrays CR and CC respectively of size m and n correspondingly containing the number on the grid's left side and top side. If one of the arrays contains -1, then the puzzle does not have complete numerical constraints and thus we skip this verification. Otherwise, we can independently sum all the entries of CC and CR and check whether these sums are equal. We use a Boolean valued function ISSUMEQUAL(CC, CR) to carry out this algorithm. This function returns true if and only if: (1) the sum of all entries in CC and CR are identical and each array does not contain -1, or (2) either CC or CR contain -1. The analysis of the asymptotic running time of ISSUMEQUAL is as follows. Notice that finding the sum of all entries in CC and CR can be done in $O(n)$ and $O(m)$ time, respectively. Therefore, since finding the sum of entries in the array can be done separately, the asymptotic running time of ISSUMEQUAL is $O(\max\{n, m\})$.

B. Checking the Colors of All Cells in the Same Tile

To check the second rule, we make two one-indexed arrays $Colored$ and $CellTotal$ of integers whose lengths are identical to the number of tiles in the instance. Suppose we consider an instance with p tiles $T_1, T_2, \ldots, T_p$. For any $1 \leq k \leq p$, we define $Colored[k]$ as the number of colored cells in T_k. Similarly, we define $CellTotal[k]$ as the number of all cells in T_k. Notice that a configuration satisfies this rule if either $Colored[k] = 0$ or $Colored[k] = CellTotal[k]$ for any $1 \leq k \leq p$. To obtain the array $Colored$ and $CellTotal$, we visit each cell in the instance in row-major order. For each visited cell, we increase the value of $CellTotal[k]$ where k is the tile number of the visited cell. Simultaneously, we increase the value of $Colored[k]$ if the current visited cell that belongs to tile k is colored. The algorithm returns true if and only if $Colored[k] = 0$ or $Colored[k] = CellTotal[k]$ for any tile T_k ($1 \leq k \leq p$). This process is carried out using the function ISALLSAME that takes a Tilepaint configuration CG of size $m \times n$ with p tiles and it is described in Algorithm 1.

Since the function ISALLSAME needs to visit $m \times n$ cells one by one and $p \leq mn$, the asymptotic running time complexity of this function is $O(mn)$. Furthermore, we need to store a two-dimensional array CG of size $m \times n$ and the two one-dimensional arrays $Colored$ and $CellTotal$ of size $p \leq mn$, then the asymptotic space complexity of this algorithm is bounded above by $O(mn)$.

C. Checking the Number of Colored Cells for Each Row and Column

Suppose we consider a Tilepaint instance of size $m \times n$ whose rows and columns constraints are correspondingly stored in arrays CR of size m and CC of size n. To check whether a Tilepaint configuration CG of size $m \times n$ satisfies rule the third rule, we firstly define $ColoredRow$ as a one-indexed array of size m such that $ColoredRow[i]$ denotes the number of colored cells within row i in CG where $1 \leq i \leq m$. Similarly, we define $ColoredCol$ as a one-indexed array of size n such that $ColoredCol[j]$ denotes the number of colored cells within column j in CG where $1 \leq j \leq n$. Then, we do the following verification for all $1 \leq i \leq m$ and $1 \leq j \leq n$:

1) if $CR[i] \neq -1$, then the value of $ColoredRow[i]$ must be equal to $CR[i]$,
2) if $CC[j] \neq -1$, then the value of $ColoredCol[j]$ must be equal to $CC[j]$.

We can fill the arrays $ColoredRow$ and $ColoredCol$ simultaneously by visiting each cell in CG in row-major order. Notice that an (i, j) cell of CG contains a pair $(T_{i,j}, C_{i,j})$ where $C_{i,j}$ is either 0 or 1. If $C_{i,j} = 1$, we increment each of $ColoredRow[i]$ and $ColoredCol[j]$ by 1. This process is explained in Algorithm 2.

Algorithm 1 IsAllSame(CG, p) checks whether all cells in the same tile are either all colored or uncolored in a Tilepaint configuration CG of size $m \times n$ containing p tiles.

Require: A Tilepaint configuration represented in a two-dimensional array CG of size $m \times n$ containing p tiles. The (i, j) entry of CG is $(T_{i,j}, C_{i,j})$ where $T_{i,j}$ denotes the tile number (an integer between 1 and p, inclusive) and $C_{i,j}$ is either 0 or 1 where 1 represents that cell (i, j) is colored.

Ensure: The function returns true if and only if all cells in the same tile are colored or not colored; otherwise, it returns false.

```
1:  $Colored \leftarrow$ array of zeros of length $p$     $\triangleright$ stores the number of colored cells in each tile $k$
2:  $CellTotal \leftarrow$ array of zeros of length $p$     $\triangleright$ stores the total number of cells in each tile $k$
3:  for $i \leftarrow 1$ to $m$ do
4:      for $j \leftarrow 1$ to $n$ do
5:          $CellTotal[T_{i,j}] \leftarrow CellTotal[T_{i,j}] + 1$
6:          if $C_{i,j} = 1$ then     $\triangleright$ cell $(i, j)$ is colored
7:              $Colored[T_{i,j}] \leftarrow Colored[T_{i,j}] + 1$
8:          end if
9:      end for
10: end for
11: $AllSame \leftarrow$ true     $\triangleright$ stores the coloring status of all cells in each tile
12: $k \leftarrow 1$
13: while $k \leq p$ and $AllSame$ do
14:     if $Colored[k] \neq CellTotal[k]$ and $Colored[k] \neq 0$ then
15:         $AllSame \leftarrow$ false     $\triangleright$ some, but not all, cells in $T_k$ are colored
16:     end if
17:     $k \leftarrow k + 1$
18: end while
19: return $AllSame$
```

Since the function ComplyConstraint needs to visit all mn cells, the asymptotic running time complexity of this function is $O(mn)$. Furthermore, we have CG of size $m \times n$, two one-dimensional arrays CC of size n and CR of size m, and two one-dimensional arrays $ColoredCol$ of size n and $ColoredRow$ of size m. Thus, the asymptotic space complexity of this algorithm is $O(mn)$.

D. Main Verification Algorithm and Its Analysis

Suppose we are given a Tilepaint configuration of size $m \times n$ containing p of tiles represented by a two-dimensional array CG. This configuration also considers two one-dimensional arrays CC and CR, respectively representing the number column and row constraint. To verify whether the configuration CG is also a valid solution, we check whether all the following functions return true:

1) IsSumEqual(CC, CR) (the sum of all entries in CC and CR are identical or the puzzle does not have complete information),
2) IsAllSame(CG, p) (all cells in the same tile of Tilepaint configuration CG containing p tiles are either colored or uncolored),
3) ComplyConstraint(CG, CC, CR) (the Tilepaint configuration CG complies with the rows and columns contains CR and CC).

The aforementioned process is carried out using the function IsVerified(CG, CC, CR, p). Notice that the asymptotic running time complexities of the functions IsSumEqual(CC, CR), IsAllSame(CG, p), ComplyConstraint(CG, CC, CR) are respectively $O(\max\{n, m\})$, $O(mn)$, and $O(mn)$. Thus the asymptotic upper bound for the running time of IsVerified(CG, CC, CR, p) is $O(\max\{n, m\}) + O(mn) + O(mn) = O(mn)$.

Algorithm 2 ComplyConstraint(CG, CC, CR) checks whether Tilepaint configuration CG complies with the column and row constraints described in CC and CR.

```
Require: The $(i,j)$ entry of $CG$ is $(T_{i,j}, C_{i,j})$ where $T_{i,j}$ denotes the tile number (an integer
    between 1 and $p$, inclusive) and $C_{i,j}$ is either 0 or 1 with 1 represents that cell $(i,j)$ is colored.
    Two arrays $CC$ and $CR$ of sizes $n$ and $m$ respectively denote the column and row constraints
    of a Tilepaint puzzle.
Ensure: The function returns true if the Tilepaint configuration $CG$ complies with the rows and
    columns constraints $CR$ and $CC$. Otherwise, it returns false.
 1: $ColoredRow \leftarrow$ array of zeros of length $m$
 2: $ColoredCol \leftarrow$ array of zeros of length $n$
 3: for $i \leftarrow 1$ to $m$ do
 4:     for $j \leftarrow 1$ to $n$ do
 5:         if $C_{i,j} = 1$ then                                  ▷ cell $(i,j)$ is colored
 6:             $ColoredCol[j] \leftarrow ColoredCol[j] + 1$
 7:             $ColoredRow[i] \leftarrow ColoredRow[i] + 1$
 8:         end if
 9:     end for
10: end for
11: $IsComply \leftarrow$ true  ▷ the compliance status of the colored cells in the grid to the constraint
12: $j \leftarrow 1$
13: while $j \leq n$ and $IsComply$ do
14:     if $CC[j] \neq -1$ and $ColoredCol[j] \neq CC[j]$ then
15:         $IsComply \leftarrow$ false
16:         ▷ the number of colored cells in column $j$ does not comply with the constraint number
17:     end if
18:     $j \leftarrow j + 1$
19: end while
20: $i \leftarrow 1$
21: while $i \leq m$ and $IsComply$ do
22:     if $CR[i] \neq -1$ and $ColoredRow[i] \neq CR[i]$ then
23:         $IsComply \leftarrow$ false
24:         ▷ the number of colored cells in row $i$ does not comply with the constraint number
25:     end if
26:     $i \leftarrow i + 1$
27: end while
28: return $IsComply$
```

IV. Complete Search Method for Finding Tilepaint Solutions

A complete search approach can be used to solve a Tilepaint puzzle. Here, we discuss a complete search algorithm using a bitmask technique to generate all possible combinations of tiles' coloring. If there are p tiles in the instance, then there are 2^p combinations of tiles' coloring for such an instance. Suppose we have a configuration of Tilepaint instance containing p tiles. We map the configuration to an integer λ whose binary representation is represented by $(\lambda_p \lambda_{p-1} \ldots \lambda_2 \lambda_1)_2$ where $\lambda_i \in \{0, 1\}$ $(1 \leq i \leq p)$. Here, $\lambda_i = 1$ if and only if the tile T_i is colored (that is, every cell in T_i is colored). The complete search approach with the bitmask technique works as follows:

1) First, we check whether the first rule of the verification steps is satisfied. If this rule is not satisfied, the algorithm terminates and returns no solution.
2) We map all possible combinations of tiles' coloring to a binary value between 0 and $2^p - 1$ (inclusive). Suppose we have integer λ whose value is between 0 to $2^p - 1$. We have $\lambda_i = 1$ if and only if tile T_i is colored.
3) In every iteration, we increment the value of λ and color the tiles in the instance according

to the bit of λ. To check whether the i-th less significant bit of $\lambda = (\lambda_p\lambda_{p-1}\ldots\lambda_2\lambda_1)_2$ is 1, we use the *bit-wise and operation* to compute $(\lambda)_2\&(2^i)_2$ where $\&$ denotes the *bit-wise and operator* and $(\lambda)_2$ and $(2^i)_2$ denote the binary representations of λ and 2^i, respectively. If the result of this operation is non-zero, then we color tile T_i. The coloring process is performed using COLOR procedure explained in Algorithm 3.

4) For each iteration, we verify whether the configuration corresponds to λ is also a solution to the instance using ISVERIFIED function explained in Section III-D.

The above steps are explained further in Algorithm 4. To color a particular tile, we define a procedure COLOR(CG, t, v) to color the all cells in the tile T_t $(1 \le t \le p)$ with color $v \in \{0, 1\}$. This procedure is explained in Algorithm 3. The asymptotic running time of Algorithm 3 is $O(mn)$ due to the doubly-nested loop in lines 1-7.

Algorithm 3 COLOR(CG, t, v) colors all cells in the tile T_t with color v. Here $1 \le t \le p$ and $v \in \{0, 1\}$ where p denotes the number of tiles in the instance.

```
Require: The $(i,j)$ entry of $CG$ is $(T_{i,j}, C_{i,j})$ where $T_{i,j}$ denotes the tile number (an integer
    between 1 and $p$, inclusive) and $C_{i,j}$ is either 0 or 1 with 1 represents that cell $(i,j)$ is colored.
Ensure: Every cell $(i,j)$ in tile $T_{i,j}$ satisfies $C_{i,j} = v$.
1: for $i \leftarrow 1$ to $m$ do
2:     for $j \leftarrow 1$ to $n$ do
3:         if $T_{i,j} = t$ then
4:             $C_{i,j} \leftarrow v$
5:         end if
6:     end for
7: end for
```

We use the function FINDSOLUTIONEXHAUSTIVE as explained in Algorithm 4 for finding one solution to a Tilepaint instance of size $m \times n$ with p tiles represented as IG with numerical constraints CC and CR. Initially, this algorithm checks the compliance of the numerical constraints for rows and columns using ISSUMEQUAL function. Subsequently, the algorithm sets the configuration CG as IG and performs the next three aforementioned steps. If the Tilepaint instance has a solution, then Algorithm 4 returns a subset of $\{1, 2, \ldots, p\}$ as the set of colored tiles; otherwise, it returns the string "no solution".

We provide an example related to the coloring process of Algorithm 4 in a 4×4 Tilepaint instance containing four tiles in Fig. 5. The numerical constraints for this instance are $CR = [3, -1, 2, 2]$ and $CC = [4, -1, 2, 0]$. The initial instance is depicted in Fig. 5a and corresponds to $\lambda = 0 = (0000)_2$. By incrementing the value of λ by 1, we obtain the configuration in Fig. 5b that corresponds to $\lambda = 1 = (0001)_2$. Here, only tile T_1 is colored. Fig. 5c illustrates the condition after λ is incremented to $2 = (0010)_2$. Notice that only tile T_2 is colored. Finally, Fig. 5d depicts the condition after λ is incremented to $3 = (0011)_2$. This value of λ corresponds to the condition when tiles T_1 and T_2 are colored. Notice that one of the solutions to the instance in Fig. 5a is obtained when tiles T_1 and T_3 are colored, which corresponds to the value $\lambda = 5 = (0101)_2$. In this case, the algorithm terminates after five iterations. In general, the asymptotic running time for Algorithm 4 is discussed in Theorem 4.

Theorem 4. *The asymptotic upper bound for the running time of Algorithm 4 for an arbitrary Tilepaint instance of size $m \times n$ with p tiles where $1 \le p \le mn$ is $O(2^p \cdot p \cdot mn)$.*

Proof. From the aforementioned analyses of the first verification rule, the asymptotic upper bound for line 1 of Algorithm 4 is $O(\max\{m, n\})$. Observe the doubly-nested loop in lines 6-23. The inner loop in lines 8-13 and 17-21 perform p iterations each, where each iteration runs the function COLOR whose asymptotic running time upper bound is $O(mn)$. Thus, the asymptotic running time complexities of lines 8-13 and 17-21 are $O(p\cdot mn)$. Notice that the asymptotic running time upper bound for line 14 is $O(mn)$. Therefore the asymptotic running time complexity of lines 8-22 is $O(p \cdot mn) + O(mn) + O(p \cdot mn) = O(p \cdot mn)$. Since the maximum number of iterations of

Algorithm 4 FindSolutionExhaustive(IG, CC, CR, p) generates all 2^p configurations of a Tilepaint instance IG containing p tiles and checks whether one of such configurations is a solution to the instance.

Require: The (i, j) entry of IG is $(T_{i,j}, 0)$ where $T_{i,j}$ denotes the tile number (an integer between 1 and p, inclusive). Array CC containing n integers denotes the column constraint. Similarly, array CR containing m integers denotes the row constraint.

Ensure: The function returns a subset of $\{1, 2, \ldots, p\}$ representing the colored tiles (if any solution exists); otherwise, it returns the string "no solution".

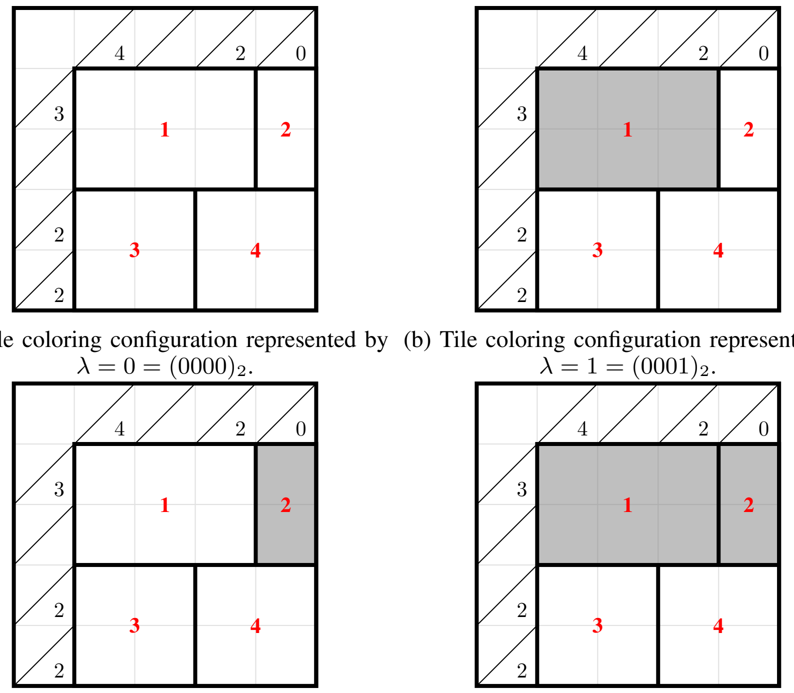
```
1: if not IsSumEqual($CC, CR$) then
2:     return "no solution", terminate the algorithm
3:     ▷ the instance has complete information but $\sum_{j=1}^{n} CC[j] \neq \sum_{i=1}^{m} CR[i]$
4: end if
5: $CG \leftarrow IG$                         ▷ initially, the configuration is the instance
6: for $i \leftarrow 0$ to $2^p - 1$ do                ▷ for generating all $2^p$ configurations
7:     $tile \leftarrow \emptyset$          ▷ stores the set of colored tiles, initially it is an empty set
8:     for $j \leftarrow 0$ to $p - 1$ do                ▷ for generating the numbers $2^j$
9:         if $(i)_2 \& (2^j)_2 \neq 0$ then           ▷ $j$-th less significant bit of $i$ is equal to 1
10:            Color($CG, j + 1, 1$)                    ▷ set all cells in $T_{j+1}$ as colored
11:            $tile.add(j + 1)$                 ▷ add tile number $j + 1$ to the solution set
12:        end if
13:    end for
14:    if IsVerified($CG, CC, CR, p$) then           ▷ the configuration $CG$ is also a solution
15:        return $tile$, terminate the algorithm
16:    else
17:        for $j \leftarrow 0$ to $p - 1$ do
18:            if $(i)_2 \& (2^j)_2 \neq 0$ then                      ▷ tile $j + 1$ is colored
19:                Color($CG, j + 1, 0$)                ▷ revert tile $j + 1$ to uncolored
20:            end if
21:        end for
22:    end if
23: end for
24: return "no solution", terminate the algorithm
```

lines 6-23 is 2^p, then we conclude that the asymptotic running time complexity of Algorithm 4 is $O(2^p \cdot p \cdot mn)$. □

Theorem 4 infers that the asymptotic running time complexity for Algorithm 4 is exponentially proportional to the number of tiles in the instance. If there is only one tile, then the running time of Algorithm 4 becomes $O(mn)$. This is because there are only two possible configurations (namely, all cells are colored or uncolored) and each configuration needs to be verified using algorithm IsVerified whose running time is $O(mn)$. On the other hand, if there are mn tiles (all tiles are of the size 1×1), then the running time of Algorithm 4 becomes $O(2^{mn} \cdot m^2 n^2)$. This is because each cell can be either colored or uncolored and there are 2^{mn} coloring configurations for such instance.

V. Prune-and-Search Method for Finding Tilepaint Solutions

Notice that Algorithm 4 generates all possible configurations without considering the observation in Theorem 2 and Corollary 1. This makes the algorithm considers many obviously invalid configurations. This section discusses a backtracking approach for solving the Tilepaint puzzle that is more efficient than the complete search algorithm. Backtracking is an algorithmic technique



(a) Tile coloring configuration represented by $\lambda = 0 = (0000)_2$.

(b) Tile coloring configuration represented by $\lambda = 1 = (0001)_2$.

(c) Tile coloring configuration represented by $\lambda = 2 = (0010)_2$.

(d) Tile coloring configuration represented by $\lambda = 3 = (0011)_2$.

Fig. 5: Some configurations related to the coloring process in Algorithm 4. The initial instance is depicted in Fig. 5a.

for finding all possible answers by trying some sequence of decisions until one finds a solution or the decision sequence is not fulfilled [29]. Moreover, the backtracking algorithm can be further optimized with the pruning technique, i.e., eliminating sequences of decisions that do not lead to answers. Our proposed backtracking algorithm considers additional pruning to solve the Tilepaint puzzle, hence turning it into a prune-and-search algorithm.

We apply Theorem 2 and Corollary 1 to the prune-and-search algorithm. Recall that if CR_i (resp. CC_j) is odd, then an odd number of odd-sized tile segments in row i (resp. column j) must be colored. Suppose we have determined some tiles' coloring status in a backtracking algorithm phase. The pruning is conducted by considering the number of the remaining odd-sized tile segments whose colors are still undetermined in each row i. Specifically, we check if the following condition holds for each row i: (1) the parity of the constraint number is different from the number of cells already colored, and (2) there are no more odd-sized tile segments left to color in row i. This condition implies that we cannot alter the parity of the number of colored cells to match the constraint number. If this condition is satisfied for a row, then the constraint number rule for that particular row is impossible to be fulfilled; thus, we prune the search space and backtrack. We may also apply the same principle when considering a column j and its numerical constraint CC_j.

In our proposed backtracking approach, we determine the color of tiles one by one from tile T_1 to T_p. We first define some variables and functions to implement the pruning step for the backtracking algorithm. We define now where $1 \leq now \leq p$ as the tile number whose coloring status is currently being determined in the algorithm. We also define $RowCell[i][T_{i,j}]$ as the number of cells for tile segments of tile $T_{i,j}$ appearing in row i. Moreover, we also define an auxiliary Boolean-valued function $\textsc{IsInUndeterminedOdd}(i, j, now)$ to check if a cell (i, j) belongs to an odd-sized tile segment in row i whose coloring status is still undetermined. This function returns true if and only if:

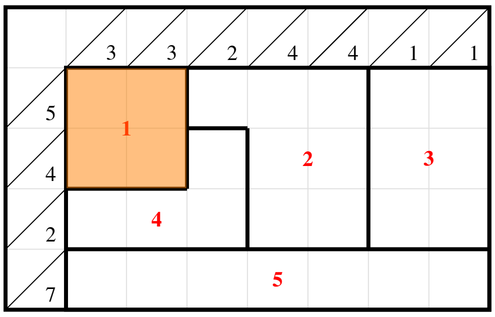
1) cell (i, j) is in an odd-sized tile segment appearing in row i (i.e., $RowCell[i][T_{i,j}]$ is odd),

2) the coloring status of the current tile that contains (i, j) is still undetermined (i.e., $T_{i,j} > now$).

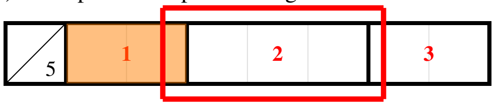
The algorithm works as follows. Initially, we define two arrays $ColoredRow$ and $ColoredCol$ respectively of size m and n such that $ColoredRow[i]$ and $ColoredCol[j]$ correspondingly denote the numbers of colored cells in row i and column j where $1 \le i \le m$ and $1 \le j \le n$. For each row i, we do the aforementioned examination as follows:

1) Suppose we consider a row i where $1 \le i \le m$. Firstly, we check whether row i has a constraint. If it has no constraint (i.e., $CR[i] = -1$), then we skip the checking for row i.
2) Otherwise, we determine the following information:
   a) $ColoredRow[i]$, which represents the number of colored cells in row i so far;
   b) $RowCell[i][k]$, which represents the number of cells in tile T_k that appear in row i (i.e., size of the tile segment of tile T_k in row i), for each $1 \le k \le p$;
   c) $OddRowRemain[i]$, which represents the number of undetermined odd-sized tile segments in row i; we compute this utilizing the IsInUndeterminedOdd function.
3) Finally, we check if the parity of $ColoredRow[i]$ is different from the constraint number $CR[i]$, and $OddRowRemain[i] = 0$. If it is the case, then we must prune the search space and backtrack.

We provide an example of the above process in a 4×7 Tilepaint instance in Fig. 6. Suppose we are examining row 1 in the Tilepaint instance depicted in Fig. 6a (the first row is also depicted in Fig. 6b). Observe that $ColoredRow[1]$ is even and $CR[1]$ is odd. Therefore, we must color an undetermined odd-sized tile segment to change the parity of $ColoredRow[1]$. We first compute the previously mentioned information to check if this is possible. We have $OddRowRemain[1] = 1$ (from T_2), indicating that we can still change the parity of the number of colored cells for row 1 by coloring T_2. If this were not the case, no additional coloring would assist in solving the puzzle, and we would need to backtrack. Analogously, the same process can be done for the column constraints. This examination process is summarized in Algorithm 5 defined by the Boolean valued function IsRemainRowUncolored.



(a) Example of Tilepaint configuration of size 4×7.



(b) Row 1 of Fig. 6a, tile number 2 should be colored.

Fig. 6: Determining possible coloring for cells with odd row constraint with even number of colored cells so far.

Since Algorithm 5 needs to visit $m \times n$ cells one by one, then the asymptotic running time complexity is $O(mn)$. We have CG of size $m \times n$, a one-dimensional array CR of size m, a one-dimensional array $ColoredRow$ of size m, a one-dimensional array $OddRowRemain$ of size m, and $RowCell$ of size $m \times p$. Therefore, the asymptotic space complexity for this algorithm is $O(\max\{mn, mp\})$ or $O(m \cdot \max\{n, p\})$. Similarly, to check for the compliance of the column constraints, we define a Boolean-valued function IsRemainColumnUncolored

Algorithm 5 IsRemainRowUncolored(CG, CR, now) checks if the configuration CG is a possible solution by counting the remaining number of undetermined odd-sized tiles segment in each row by considering tile T_{now}.

Require: The (i,j) entry of CG is $(T_{i,j}, C_{i,j})$ where $T_{i,j}$ denotes the tile number (an integer between 1 and p, inclusive) and $C_{i,j}$ is either 0 or 1 with 1 represents that cell (i,j) is colored. Array CR containing m integers denotes the row constraint. A number now denotes the tile number whose coloring status is last determined.

Ensure: The function returns true if it is still possible to color tile T_{now} and for each row to match the parity of the constraint number described in CR. Otherwise, it returns false.

```
1: $ColoredRow \leftarrow$ array of zeros of length $m$
2: ▷ array to store the number of colored cells in each row
3: $OddRowRemain \leftarrow$ array of zeros of length $m$
4: ▷ array to store the number of odd-sized tile segments that remain undetermined in each row
5: $RowCell \leftarrow$ array of zeros of size $m \times p$
6: ▷ $RowCell[i][k]$ stores the number of cells in row $i$ appearing in a tile segment of $T_k$
7: $HasOddTS \leftarrow$ **true**        ▷ signify the existence of an undetermined odd-sized tile segment
8: $i \leftarrow 1$
9: **while** $i \leq m$ **and** $HasOddTS$ **do**
10:    **if** $CR[i] \neq -1$ **then**
11:        ▷ check if row $i$ has no constraint number
12:        **for** $j \leftarrow 1$ to $n$ **do**
13:            **if** $C_{i,j} = 1$ **then**        ▷ cell $(i,j)$ is colored
14:                $ColoredRow[i] \leftarrow ColoredRow[i] + 1$
15:            **end if**
16:            $RowCell[i][T_{i,j}] \leftarrow RowCell[i][T_{i,j}] + 1$
17:        **end for**
18:        **if** $ColoredRow[i]$ **mod** $2 = CR[i]$ **mod** $2$ **then**
19:            continue
20:            ▷ skip the parity checking since the parity for the number of colored cells equals
    the parity of the constraint number
21:        **end if**
22:        **for** $j \leftarrow 1$ to $n$ **do**
23:            **if** IsInUndeterminedOdd$(i, j, now)$ **then**
24:                ▷ check whether the current $RowCell[i][T_{i,j}]$ is odd, $T_{i,j} > now$, and the color
    of tile $T_{i,j}$ is still undetermined
25:                $OddRowRemain[i] \leftarrow OddRowRemain[i] + 1$
26:                $RowCell[i][T_{i,j}] \leftarrow 0$    ▷ set $RowCell[i][T_{i,j}]$ to 0 to avoid double-counting
27:            **end if**
28:        **end for**
29:        **if** ($ColoredRow[i]$ is even) **and** ($CR[i]$ is odd) **and** $OddRowRemain[i] = 0$ **then**
30:            $HasOddTS \leftarrow$ **false**
31:            ▷ $CR[i]$ is odd, no remaining odd-sized tile segment in row $i$
32:        **end if**
33:        **if** ($ColoredRow[i]$ is odd) **and** ($CR[i]$ is even) **and** $OddRowRemain[i] = 0$ **then**
34:            $HasOddTS \leftarrow$ **false**
35:            ▷ $ColoredRow[i]$ is odd, no remaining odd-sized tile segment in row $i$
36:        **end if**
37:    **end if**
38:    $i \leftarrow i + 1$
39: **end while**
40: **return** $HasOddTS$
```

that works analogously to ISREMAINROWUNCOLORED. We leave the detailed description of ISREMAINCOLUMNUNCOLORED to the readers.

Finally, the main process of the backtracking algorithm works as follows:

1) First, we check whether the sum of the row constraints and the sum of the column constraints are equal or if the puzzle has no complete information using ISSUMEQUAL function, similar to the check of the first verification rule. If this condition is not satisfied, we terminate the algorithm and conclude that the instance has no solution.
2) We determine the color of tiles one by one from tile T_1 to T_p. We assign 1 to tile T_k $(1 \leq k \leq p)$ if all cells within this tile are colored, and we assign 0 otherwise. We use the variable now to track the currently determined tile.
3) Along with the tile coloring, we check if the current configuration still satisfies the constraint of the columns and rows. Suppose we color tile T_{now}:
   a) If $ColoredRow[i] > CR[i]$ or $ColoredCol[j] > CC[j]$, then T_{now} must not be colored. The reason is the configuration no longer satisfies the constraint even if we color T_{now+1}. Therefore, we uncolor T_{now} and continue to backtrack.
   b) If the aforementioned function ISREMAINROWUNCOLORED or ISREMAINCOLUMNUNCOLORED return false, then the configuration no longer satisfies the constraint. Thus, we uncolor T_{now} and continue to backtrack.

   This process is summarized in Algorithm 6.
4) If the coloring status of all cells has been determined, a configuration is found. We use Algorithm 2 to ensure the configuration is a solution. If that is the case, we output a list (possibly empty) representing the number of colored tiles in the configuration and terminate the algorithm.

The main backtracking algorithm is explained in Algorithm 7, which takes a configuration CG, arrays of columns' and rows' constraints CC and CR, the number of tiles p, the currently investigated tile now, and a list $tile$ containing the number of all colored tiles the current state of the backtracking process. This backtracking algorithm is used for the prune-and-search algorithm for finding a solution to the puzzle as explained in Algorithm 8. This prune-and-search algorithm begins the backtracking process from the top-leftmost tile (i.e., the tile to which the cell $(1, 1)$ belongs). We illustrate a pruned state space tree generated by Algorithm 8 in solving a 2×3 Tilepaint puzzle containing four regions in Fig. 7. The following theorem discusses the asymptotic upper bound for the running time of our prune-and-search approach in solving an $m \times n$ Tilepaint instance with p tiles.

Theorem 5. *The asymptotic upper bound for the running time of Algorithm 8 for an arbitrary Tilepaint instance of size $m \times n$ with p tiles where $1 \leq p \leq mn$ is $O(2^p \cdot mn)$.*

Proof. The prune-and-search technique described in Algorithm 8 which uses the backtracking approach in Algorithm 7 has two possible states in each phase, i.e., a tile being colored or left uncolored. Thus, the steps in Algorithm 8 can be modeled with a state space tree with two possibilities for each non-terminal state. In the worst case, we try every possible tile coloring in the instance. Therefore, there are at most 2^i states at level i in the state space tree. Consequently, the number of states in this state space tree model is $\sum_{i=0}^{p} 2^i = 2^{p+1} - 1$. Moreover, we have a function COLOR to color tiles and a function COMPLYCONSTRAINT to verify the compliance of the resulting configuration in each state, in which the running time complexity of both functions is $O(mn)$. As a result, the computational cost of Algorithm 8 is bounded above by $O(2^p \cdot mn)$. $\square$

Theorem 5 infers that the asymptotic running time for Algorithm 8 is exponentially proportional to the number of tiles in the instance. Nevertheless, Algorithm 8 is asymptotically faster than the complete search method explained in Algorithm 4 by a factor of p. If an $m \times n$ Tilepaint instance has only one tile, the running time of Algorithm 8 becomes $O(mn)$, which is similar to that of Algorithm 4 for the same case. However, if such an instance has mn tiles of size 1×1, the running time of Algorithm 8 becomes $O(2^{mn} \cdot mn)$, which is asymptotically faster than Algorithm 4 by a factor of mn.

Algorithm 6 ISVALIDCONF(CG, CC, CR, now) returns true if the configuration CG with column and row constraints CC and CR is still possible to comply with the puzzle's rule after tile T_{now} is colored.

```
Require: The $(i, j)$ entry of $CG$ is $(T_{i,j}, C_{i,j})$ where $T_{i,j}$ denotes the tile number (an integer
    between 1 and $p$, inclusive) and $C_{i,j}$ is either 0 or 1 with 1 represents that cell $(i, j)$ is
    colored. Array $CC$ containing $n$ integers denotes the column constraint. Similarly, array $CR$
    containing $m$ integers denotes the row constraint. A number $now$ denotes the tile number
    whose coloring status was last determined.
Ensure: The function returns true if the configuration is still possible to comply with the puzzle's
    rule after $T_{now}$ is colored. Otherwise, it returns false.
 1: $ColoredRow \leftarrow$ array of zeros of length $m$
 2: $ColoredCol \leftarrow$ array of zeros of length $n$
 3: for $i \leftarrow 1$ to $m$ do
 4:     for $j \leftarrow 1$ to $n$ do
 5:         if $C_{i,j} = 1$ then                                  ▷ cell $(i, j)$ is colored
 6:             $ColoredRow[i] \leftarrow ColoredRow[i] + 1$
 7:             $ColoredCol[j] \leftarrow ColoredCol[j] + 1$
 8:         end if
 9:     end for
10: end for
11: $NotExeceedCRCC \leftarrow$ true
12: ▷ signify that the number of colored cells is not more than the corresponding constraint
13: $j \leftarrow 1$
14: while $j \leq n$ and $NotExeceedCRCC$ do
15:     if $CC[j] \neq -1$ and $ColoredCol[j] > CC[j]$ then
16:         $NotExeceedCRCC \leftarrow$ false
17:         ▷ more colored cells in column $j$ than its column's constraint
18:     end if
19:     $j \leftarrow j + 1$
20: end while
21: $i \leftarrow 1$
22: while $i \leq m$ and $NotExeceedCRCC$ do
23:     if $CR[i] \neq -1$ and $ColoredRow[i] > CR[i]$ then
24:         $NotExeceedCRCC \leftarrow$ false
25:         ▷ more colored cells in row $i$ than its row's constraint
26:     end if
27:     $i \leftarrow i + 1$
28: end while
29: $RemainRow \leftarrow$ ISREMAINROWUNCOLORED($CG, CR, now$)
30: $RemainCol \leftarrow$ ISREMAINCOLUMNUNCOLORED($CG, CC, now$)
31: return $NotExeceedCRCC$ and $RemainRow$ and $RemainCol$
```

VI. TRACTABLE VARIANTS OF TILEPAINT PUZZLES

In some cases, NP-complete problems may contain subproblems or specific cases that belong to the class P, which consists of problems solvable in polynomial time. For example, the general k-CNF-SAT is NP-complete, but the 2-CNF-SAT is solvable in linear time by means of implication graphs [30]. The general Nonogram puzzle, which is related to two-dimensional discrete tomography, is NP-complete, but the problem is solvable in polynomial time if every row and column contains a single block of connected cells [31]. Finally, although the general Yin-Yang puzzle is NP-complete, its variant of size $m \times n$ where m or n is less than 3 is solvable in linear time [20, Theorem 2 and Theorem 3].

Algorithm 7 BACKTRACK($now, CG, p, CC, CR, tile$) determines the coloring status of the current tile number now recursively in a configuration CG containing p tiles with column and row constraints CC and CR and storing the current configuration of colored tiles in a list $tile$.

Require: The (i, j) entry of CG is $(T_{i,j}, C_{i,j})$ where $T_{i,j}$ denotes the tile number (an integer between 1 and p, inclusive) and $C_{i,j}$ is either 0 or 1 with 1 represents that cell (i, j) is colored. Array CC size n of integers denotes the column constraint. Similarly, array CR size m of integers denotes the row constraint. A list $tile$ of tile numbers that are colored in the configuration. The backtracking algorithm currently investigates the coloring for tile now ($1 \leq now \leq p$).

Ensure: The procedure updates $tile$ as the list of the colored tiles' numbers representing the possible solution to the instance.

```
1: if $now > p$ then                                    ▷ no more tile coloring status to determine
2:     if COMPLYCONSTRAINT($CG, CC, CR$) then
3:         output($tile$)        ▷ output the configuration, which is also a solution to the instance
4:         terminate the procedure                  ▷ procedure terminates if a solution is found
5:     end if
6: else
7:     for $i \in \{1, 0\}$ do                              ▷ two status of tile's color to determine
8:         COLOR($CG, now, i$)                                    ▷ color all cell in $T_{now}$ by $i$
9:         if $i = 0$ then                                ▷ $T_{now}$ colored by 0, i.e., left white
10:            BACKTRACK($now + 1, CG, p, CC, CR, tile$)                 ▷ continue to the next tile
11:        else                                            ▷ $T_{now}$ colored by 1, i.e., blackened
12:            $tile.append(now)$                               ▷ add the $now$ to the list $tile$
13:            if not ISVALIDCONF($CG, CC, CR, now$) then
14:                ▷ the configuration cannot possibly complies with the constraint
15:                COLOR($CG, now, 0$)                             ▷ revert $T_{now}$ to uncolored
16:                $tile.pop\_back()$                    ▷ take out the last stored entry in $tile$
17:            else
18:                BACKTRACK($now + 1, CG, p, CC, CR, tile$)             ▷ continue to the next tile
19:                $tile.pop\_back()$                    ▷ take out the last stored entry in $tile$
20:                COLOR($CG, now, 0$)                             ▷ revert $T_{now}$ to uncolored
21:            end if
22:        end if
23:    end for
24: end if
```

Algorithm 8 FINDSOLUTIONPRUNE(CG, p, CC, CR) is the main prune-and-search algorithm to find the solution from configuration CG containing p tiles with column and row constraint CC and CR.

Require: The (i, j) entry of CG is $(T_{i,j}, C_{i,j})$ where $T_{i,j}$ denotes the tile number (an integer between 1 and p, inclusive) and $C_{i,j}$ is either 0 or 1 with 1 represents that cell (i, j) is colored. Array CC size n of integers denotes the column constraint. Similarly, array CR size m of integers denotes the row constraint. The number of tiles in the instance is denoted by p.

Ensure: The procedure returns the list of colored tiles' numbers representing the solution to the instance (if any) or a string "no solution" otherwise.

```
1: $tile \leftarrow$ empty list
2: if not ISSUMEQUAL($CR, CC$) then
3:     return "no solution"
4:     ▷ the instance has complete information but $\sum_{j=1}^{n} CC[j] \neq \sum_{i=1}^{m} CR[i]$
5: end if
6: BACKTRACK($1, CG, p, CC, CR, tile$)
7: ▷ start the backtracking process from the top-leftmost region to which the cell $(1, 1)$ belongs
```

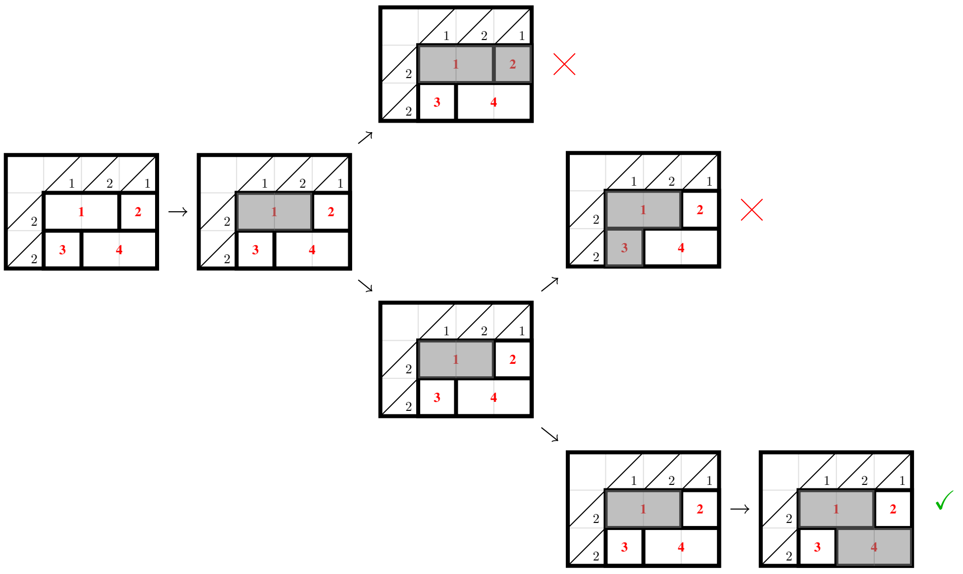


Fig. 7: The pruned state space tree generated by Algorithm 8 in solving a 2×3 Tilepaint instance with four tiles (far left). The i-th level of the tree determines whether the tile i is colored or uncolored. The grid states that are marked with red crosses are the state that cannot possibly lead to a solution (i.e., it does not satisfy one of the conditions in step 3), while the grid with a green check mark is the solution.

In this section, we discuss the tractable variants of Tilepaint puzzles of size $m \times n$ with mn tiles of size 1×1 and complete information (i.e., the constraints for all rows and columns are defined). Notice that, according to Theorem 5, solving such an instance using the prune-and-search approach as in Algorithm 8 requires $O(2^{mn} \cdot mn)$ time, which is exponential in terms of the puzzle's size. However, this instance resembles a two-dimensional discrete tomography problem with complete information, which has been extensively studied by Ryser [24] and Herman and Kuba [25]. We refer to this instance as an "mn-tile instance" for brevity.

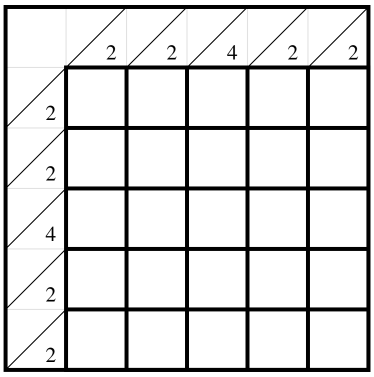
In this section, we treat the mn-tile instance as an instance of a two-dimensional discrete tomography problem. Using the techniques in [24], [25], we show that such an instance is solvable in polynomial time. Fig. 8 describes an mn-tile instance and its two-dimensional discrete tomography counterpart.

A. Verifying the Solution's Existence of mn-tile Instance

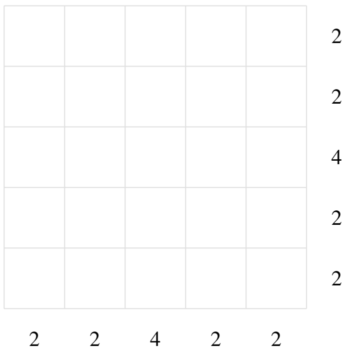
Given an mn-tile instance, we can verify whether such an instance has a solution in a polynomial time. As in Section II, suppose the constraints for the rows and columns are respectively denoted by $CR = [CR_1, CR_2, \ldots, CR_m]$ and $CC = [CC_1, CC_2, \ldots, CC_n]$. Based on [24], we can verify the existence of the solution to this instance using the following steps:

1) We verify if the sum of the constraint number for the rows and columns is identical as described in Theorem 1.
2) We verify that $CR_i \leq n$ and $CC_j \leq m$ for all $1 \leq i \leq m$ and $1 \leq j \leq n$.
3) Given $X = [x_1, x_2, \ldots, x_n]$ where $x_j = |\{i : CR_i \geq j\}|$ and $1 \leq j \leq n$, then X must *majorize* CC' which is obtained from sorting the entries of CC in non-increasing order. It can be shown that X is already sorted in non-increasing order. Let $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_n]$ are two arrays whose entries are sorted in non-increasing order, then A majorizes B if $\sum_{i=1}^{k} a_i \geq \sum_{j=1}^{k} b_j$ for every $1 \leq k \leq n$.

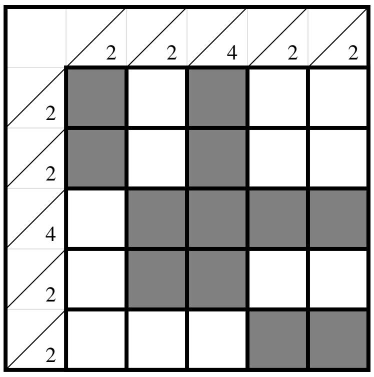
We can easily perform the first two steps in $O(\max\{m, n\})$ time. As for the third rule related to majorization, the verification process can be done by computing the prefix sum of arrays of X and



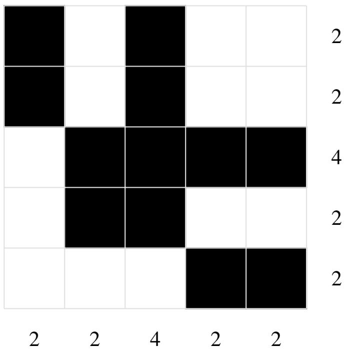
(a) An mn-tile instance with complete information of size 5×5.



(b) A two-dimensional discrete tomography analog to the instance in Fig. 8a.



(c) A solution to the instance in Fig. 8a.



(d) A solution to the instance in Fig. 8b.

Fig. 8: An mn-tile instance and its corresponding two-dimensional discrete tomography problem counterpart.

CC'. Notice that the construction of X can be done in $O(mn)$ time. Recall that the prefix sum array of X is defined as PX where $PX[i] = \sum_{k=1}^{i} X[k]$. Similarly, the prefix sum array of CC' is defined as PCC where $PCC[i] = \sum_{k=1}^{i} CC'[k]$. We need to check that $PX[i] \geq PCC[i]$ for all $1 \leq i \leq n$. The construction of both PX and PCC can be performed $O(n)$ time. Since an optimal comparison-based sorting of n entries requires $O(n \log n)$ time, then the overall process in the third step requires $O(n \cdot \max\{m, \log n\})$ time. However, if we instead use a linear time sorting technique, such as counting sort for constructing CC', the running time of the third step becomes $O(mn)$.

For future reference, let us define IsMNTileVerified(CR, CC, m, n) as a function that returns true if and only if an mn-tile instance with the row and column constraints CR and CC complies with all three aforementioned rules. In other words, IsMNTileVerified(CR, CC, m, n) returns true if and only if such an instance has a solution. Based on the aforementioned analyses, the asymptotic upper bound for the running time of this verification process can be expressed as $2 \cdot O(\max\{m, n\}) + O(n \cdot \max\{m, \log n\}) = O(n \cdot \max\{m, \log n\})$, that is, we can check whether an mn-tile instance has a solution in $O(n \cdot \max\{m, \log n\})$ time. Notice that if we use a linear time sorting algorithm such as counting sort for sorting CC, this process can be achieved in $O(mn)$ time.

B. Solving an mn-tile Tilepaint Instance Using Greedy Algorithm

In [24], Ryser shows that we can construct a solution to an mn-tile instance by incrementally transforms X where $X = [x_1, x_2, \ldots, x_n]$, $x_j = |\{i : CR_i \geq j\}|$ into CC. Nevertheless, there is an easier and more efficient approach, as discussed by Stolk in [32, Example 1.1.15]. The idea for constructing the solution is to color the cells greedily in columns, starting with the column with the highest constraint number. For each column, we color the cells in rows that require the most colors to satisfy their constraint numbers. In each iteration, a column is considered to require the most colors if it has the largest value of row constraint minus the number of cells already colored in that column. The following steps elaborate the greedy algorithm for solving an mn-tile instance:

1) First, we check whether the instance has a solution using the function ISMNTILEVERIFIED as in Section VI-A. If the function returns false, we terminate the algorithm and conclude that the instance has no solution.
2) We construct an array Col of size n to store the order of column indices to process, starting from the column with the highest constraint number to the lowest. In other words, Col stores the permutation of the indices of CC sorted in non-increasing order.
3) We construct an array Row of size m to store the order of row indices to process, starting from the row with the highest constraint number to the lowest. In other words, Row stores the permutation of the indices of CR sorted in non-increasing order.
4) For each column index j in Col and row index r in Row, we color the cell (i, j) such that $i = Row[r]$. For each coloring action, we also decrease the value of $CR[i]$ and $CC[j]$ simultaneously.
5) Steps 3 and 4 are repeated $CC[j]$ times for each column index j $(1 \leq j \leq n)$.
6) A solution is found if the process has been performed for all column indices.

We expound this process in Algorithm 9. Observe that line 4 of Algorithm 9 constructs an array Col of length n by sorting the array CC. This process takes $O(n \log n)$ time. Notice that if we use a linear time sorting algorithm such as counting sort for constructing Col, this process can be achieved in $O(n)$ time. Consider a doubly-nested loop in lines 5-13. The inner loop in lines 7-12 performs $CC[j]$ $(1 \leq j \leq n)$ iterations where $CC[j] \leq m$. Notice that line 6 of Algorithm 9 constructs an array Row of length m by sorting the array CR. This step requires $O(m \log m)$ time. If we instead use a linear time sorting algorithm such as counting sort for constructing Row, this step can be achieved in $O(m)$ time. Since the outer loop in lines 5-13 performs n iterations, the asymptotic upper bound for lines 5-13 becomes $O(n \cdot (m \log m + m)) = O(mn \log m)$. Since line 4 takes $O(n \log n)$ time, the overall asymptotic upper bound for the running time of Algorithm 9 becomes $O(mn \log m) + O(n \log n) = O(mn \log m + n \log n)$. Using a linear time sorting algorithm makes the time complexity of Algorithm 9 becomes $O(mn)$.

VII. INTRACTABILITY OF ONE-DIMENSIONAL TILEPAINT PUZZLES

Some NP-hard puzzles remain computationally hard even if we reduce the dimension of such puzzles. One notable example is the famous video game-based puzzle Tetris, which is hard even if the number of rows or columns is bounded by a constant [33]. Suppose we restrict the Tilepaint instance to a size of $m \times 1$ or $1 \times n$, hence making it *one-dimensional*. We find that the puzzle generally remains intractable to solve even in this reduced form. This section discusses Tilepaint puzzles of this type whose formal definition is given in Definition 5.

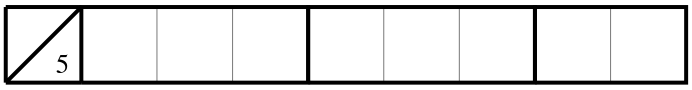
Definition 5. *A one-dimensional Tilepaint is a Tilepaint puzzle with only one row or column with a constraint number. The cells in this puzzle are divided into tiles, visually represented by bold lines. An integer is located at the top (for an instance of size* $m \times 1$*) or on the left (for an instance of size* $1 \times n$*), indicating the required number of colored cells. Each cell must be either colored or uncolored. Additionally, all cells within the same tile must share the same color, either colored or uncolored.*

Since a Tilepaint puzzle of size $m \times 1$ can be transformed into a puzzle of size $1 \times m$, we only focus on $1 \times n$ puzzle where $n \in \mathbb{N}$. An example of a 1×8 Tilepaint puzzle and its corresponding solutions is discussed in Example 2.

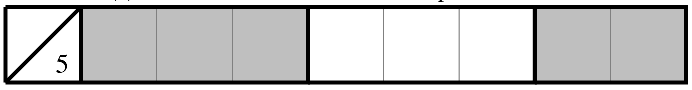
Algorithm 9 SOLVEMNTILE(CG, CR, CC, m, n) returns a configuration CG (if any) which is a solution to an mn-tile instance with row and column constraints CR and CC.

```
Require: The $(i, j)$ entry of $CG$ is $(T_{i,j}, C_{i,j})$ where $T_{i,j}$ denotes the tile number (an integer
    between 1 and $mn$, inclusive) and $C_{i,j}$ is either 0 or 1 with 1 represents that cell $(i, j)$ is
    colored. The row constraint is the array $CR$ of size $m$, while the column constraint is the
    array $CC$ of size $n$.
Ensure: The function returns the solution to the instance (if any). Otherwise, it returns a string
    "no solution".
 1: if not IsMNTileVerified$(CR, CC, m, n)$ then
 2:     return "no solution", terminate the algorithm
 3: end if
 4: $Col \leftarrow$ permutation of the indices of $CC$ sorted in non-increasing order
 5: for $j$ in $Col$ do
 6:     $Row \leftarrow$ permutation of the indices of $CR$ sorted in non-increasing order
 7:     for $r \leftarrow 1$ to $CC[j]$ do
 8:         $i \leftarrow Row[r]$
 9:         $C_{i,j} \leftarrow 1$
10:         $CC[j] \leftarrow CC[j] - 1$
11:         $CR[i] \leftarrow CR[i] - 1$
12:     end for
13: end for
14: return $CG$
```

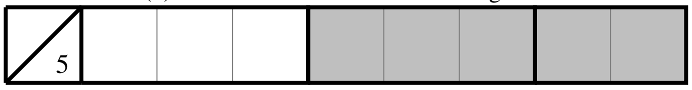
Example 2. *Suppose we consider a Tilepaint instance of size 1×8 in Fig. 9. In Fig. 9a we consider an instance with three tiles of size 3, 3, and 2 (from left to right). We must color five cells according to the value of the constraint number. There are two solutions to such an instance, depicted in Fig. 9b and Fig. 9c. It is also possible to have instances with no solution as in Fig. 9d and Fig. 9e.*



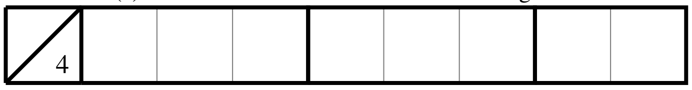
(a) A 1×8 one-dimensional Tilepaint instance.



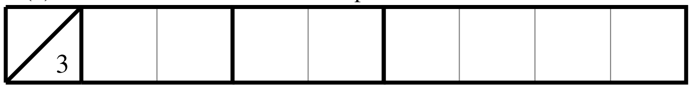
(b) A solution to the instance in Fig. 9a.



(c) Another solution to the instance in Fig. 9a.



(d) A 1×8 one-dimensional Tilepaint instance with no solution.



(e) Another 1×8 one-dimensional Tilepaint instance with no solution.

Fig. 9: Several instances and solutions of one-dimensional Tilepaint puzzles. Fig. 9a is an instance with two solutions depicted in Fig. 9b and Fig. 9c. Fig. 9d and Fig. 9e depict two distinct instances with no solution.

In a $1 \times n$ Tilepaint instance with p tiles $T_1, T_2, \ldots, T_p$, we define s_i for each $1 \leq i \leq p$ as the size (the number of cells) of tile T_i. We can represent these sizes using an array of integers $S = [s_1, s_2, \ldots, s_p]$. For example, the array representing the sizes of the tiles in the instance in Fig. 9a is $S = [3, 3, 2]$. Since the constraint number is five, we need to color five cells in this instance such that all cells in the same tile must be colored or uncolored. Notice that the condition is identical to the problem of taking some entries of S so that the sum of these entries equals to the constraint number. For the Tilepaint instance in Fig. 9a, we have $s_1 + s_3 = 5$ or $s_2 + s_3 = 5$, which correspondingly represent the solutions in Fig. 9b and Fig. 9c. Notice that we identify a connection between the one-dimensional Tilepaint puzzle and the well-known subset-sum problem. The constraint number becomes the target sum and our objective is to find a sub-array of S whose sum equals to the constraint number.

A. Important Observation for One-dimensional Tilepaint Puzzle

The theorems from Section II also apply to the one-dimensional Tilepaint puzzle. It is possible to determine whether a tile must be colored based on the constraint number's parity. We first refine Definition 3 for a one-dimensional Tilepaint puzzle in Definition 6.

Definition 6. *Suppose we consider a $1 \times n$ Tilepaint instance with p tiles. Let mo and me respectively be the number of colored tiles where the tiles' sizes are odd and even, where $0 \leq mo + me \leq p$. Let $RO_1, RO_2, \ldots, RO_{mo}$ and $RE_1, RE_2, \ldots, RE_{me}$ correspondingly denote the collection of tiles with odd and even sizes. For each RO_i, we define so_i as the sizes of RO_i. Moreover, we define se_i as the sizes of RE_i.*

The following theorem is a special case of Theorem 2 for the one-dimensional Tilepaint puzzle.

Theorem 6. *Suppose we consider a $1 \times n$ Tilepaint instance with p tiles $T_1, T_2, \ldots, T_p$. Suppose CR is the constraint number. If CR is odd, then there must be an odd number of colored tiles where each region is of an odd size.*

Proof. The proof is analogous to that of Theorem 2. □

In the following theorem, we state that if a constraint number is positive but less than the size of any tile in the instance, then such an instance has no solution.

Theorem 7. *Suppose we consider a $1 \times n$ Tilepaint instance with p tiles $T_1, T_2, \ldots, T_p$. Suppose $S = [s_1, s_2, \ldots, s_p]$ is an array where s_i is the size of T_i ($1 \leq i \leq p$). If CR satisfies $0 < CR < s_i$ for all $1 \leq i \leq p$, then the corresponding Tilepaint instance has no solution.*

Proof. The proof is analogous to that of Theorem 3. □

B. NP-Completeness of the Subset-Sum Problem

Suppose we have an array containing n integers. Given a target number $h \in \mathbb{Z}$, the subset-sum problem is a problem to determine the existence of the subset of those n numbers whose sum is exactly h. This problem is a special case of the Binary Knapsack Problem. It is well-known that the aforementioned subset-sum problem can be solved using a dynamic programming approach in $O(nh)$ pseudo-polynomial time. This is because, even though $O(nh)$ is polynomial in terms of n and h, this expression is exponential if we consider the number of bits to represent h. That is, the complexity for solving such a problem is $O(n \cdot 2^m)$ where m is the number of bits to represent h. The subset-sum problem can be reduced from the 3-SAT problem (see, e.g., [28], [34]).

C. The Reduction of the Subset-Sum Problem to the One-dimensional Tilepaint Puzzle

Firstly, we recall the definition of the subset-sum instance in Definition 7. However, we are only interested in arrays with positive integer entries and positive target sum in this definition.[2]

[2]The original subset-sum problem, which is proven NP-complete, involves an array of non-positive integers. However, reducing the 3-SAT problem to a subset-sum problem involving an array containing only positive integers and positive target sum is possible. See, e.g., [35].

Definition 7. *Suppose we consider an array $A = [a_1, a_2, \dots, a_k]$ of size k where $a_i \in \mathbb{N}$ and $h \in \mathbb{N}$. A subset-sum instance concerning A and h is defined as a pair (A, h) where h is the target sum. The solution to this instance is a sub-array of A whose sum of entries equals h. For convenience, we denote the sum of all entries in A by S.*

To construct a reduction from the subset-sum instance to the Tilepaint puzzle, we first formalize the definition of an $m \times n$ Tilepaint instance IG with numerical constraints CR and CC as a five-tuple (m, n, IG, CR, CC). Notice that a $1 \times n$ Tilepaint instance of our interest is defined as $(1, n, IG, CR, CC)$, where $CR = [CR_1]$ and CC is an array of -1 of length n. We are interested in the case where CR_1 is a positive integer since if $CR_1 = -1$ or $CR_1 = 0$, then we have a trivial solution to the instance.

Suppose we consider a subset-sum instance (A, h) where $A = [a_1, a_2, \dots, a_k]$ and $h \in \mathbb{N}$ as defined in Definition 7. We can convert this instance into a Tilepaint instance by creating k tiles whose sizes correspond to each entry of A, that is, a_i for $1 \leq i \leq k$ corresponds to the size of tile T_i. Moreover, we set h as the constraint number CR_1. We define the function $f((A, h))$ that converts a subset-sum instance (A, h) to a one-dimensional Tilepaint instance as follows:

$$\begin{aligned}
&f((A,h)) = (m, n, IG, CR, CC) \\
&m = 1, n = S \\
&IG = [[\underbrace{(1,0),\dots,(1,0)}_{a_1 \text{ terms}}, \underbrace{(2,0),\dots,(2,0)}_{a_2 \text{ terms}}, \dots, \underbrace{(k,0),\dots,(k,0)}_{a_k \text{ terms}}]] \\
&CR = [h], CC = [\underbrace{-1,\dots,-1}_{S \text{ terms}}]
\end{aligned} \tag{1}$$

The reduction steps of a subset-sum instance into its corresponding Tilepaint instance are as follows:

1) First, we define a two-dimensional array IG of size $1 \times S$ representing a one-dimensional Tilepaint instance. Each entry of IG is $(T_{1,j}, 0)$ where $T_{1,j}$ denotes the tile number (an integer between 1 and k, inclusive) and 0 represents that the cell $(1, j)$ is uncolored. Moreover, we define a number S to store the size/width of the one-dimensional Tilepaint instance. Hereafter, we can assign CR as a one-dimensional array containing the only row constraint represented by a single integer h.
2) For each a_i in A, we construct a_i cells corresponding to tile T_i. We have a_1 cells in tile T_1, a_2 cells in tile T_2, and so on up to a_k cells in tile T_k. The first a_1 entries of IG, i.e., $IG[1][j]$ where $1 \leq j \leq a_1$ are filled with $(1, 0)$, indicating that these cells belong to T_1. The next a_2 entries of IG, i.e., $IG[1][j]$ where $a_1 + 1 \leq j \leq a_1 + a_2$ are filled with $(2, 0)$, indicating that these cells belong to T_2. This process is performed until the last a_k entries of IG, i.e., $IG[1][j]$ where $S - a_k + 1 \leq j \leq S$ are filled with $(k, 0)$, indicating that these cells belong to T_k.
3) We construct a one-dimensional array CC of size S and fill each entry with -1 (since there is no column constraint).
4) Finally, the function returns a tuple of $(1, S, IG, CR, CC)$ representing the one-dimensional Tilepaint instance.

We can convert the instance of a subset-sum problem into its corresponding Tilepaint instance using Algorithm 10. Thus, the previous search-based algorithm for the Tilepaint puzzle, such as the complete search or backtracking approach, can indirectly solve the subset-sum instance in Definition 7.

We elaborate the reduction process in Algorithm 10. The inner for loop in lines 6-8 runs a_i times for each $1 \leq i \leq k$. Each step in lines 6-8 constructs a pair $(i, 0)$ for each $1 \leq i \leq k$ corresponding to the entry of cells $(1, j)$ where $(\sum_{\ell=1}^{i-1} a_\ell) + 1 \leq j \leq \sum_{\ell=1}^{i} a_\ell$, which can be done in $O(1)$ time. As a result, the loop in lines 5-10 runs $\sum_{i=1}^{k} a_i = S$ times, and thus the asymptotic running time of Algorithm 10 is $O(S)$.

In the following lemma, we prove a relation between the subset-sum and one-dimensional Tilepaint instances as described by the reduction function f in (1).

Algorithm 10 ConvertInstance(A, h) converts the subset-sum instance (A, h) where $A = [a_1, a_2, \ldots, a_k]$ with $a_i, h \in \mathbb{N}$ for $1 \leq i \leq k$ where h is the target sum into its corresponding one-dimensional Tilepaint instance.

```
Require: An array $A = [a_1, a_2, \ldots, a_k]$ and a number $h$ denoting the target sum, $a_i, h \in \mathbb{N}$ for
    every $1 \leq i \leq k$.
Ensure: A $1 \times n$ Tilepaint instance as defined in (1).
 1: $IG \leftarrow$ array of size $1 \times S$
 2: $\triangleright$ stores the corresponding one-dimensional Tilepaint instance as in (1)
 3: $PA \leftarrow 0$                                    $\triangleright$ stores the prefix sum of $A$
 4: $CR \leftarrow [h]$
 5: for $i \leftarrow 1$ to $k$ do
 6:     for $j \leftarrow 1$ to $A[i]$ do
 7:         $IG[1][PA + j] \leftarrow (i, 0)$
 8:     end for
 9:     $PA \leftarrow PA + A[i]$
10: end for
11: $CC \leftarrow$ array of size $S$
12: for $i \leftarrow 1$ to $S$ do
13:     $CC[i] \leftarrow -1$
14: end for
15: return $(1, S, IG, CR, CC)$
```

Lemma 1. *A subset-sum instance (A, h) as in Definition 7 has a solution if and only if its corresponding one-dimensional Tilepaint instance has a solution.*

Proof. Let us consider a subset-sum instance (A, h) with $A = [a_1, a_2, \ldots, a_k]$ and its corresponding one-dimensional Tilepaint instance $(1, S, IG, CR, CC)$ where $CR = [h]$ and CC is an array of -1 of length S where $S = \sum_{i=1}^{k} a_i$. Let us denote the i-th tile by T_i where $1 \leq i \leq k$. Tile T_1 contains first a_1 cells, T_2 contains the next a_2 cells, and so on up to T_k contains the last a_k cells. We split the proof in two parts as follows.

Part 1: If (A, h) has a solution, then its corresponding one-dimensional Tilepaint instance has a solution. Suppose the subset-sum instance (A, h) has a solution, which means that there is a sub-array of A whose sum equals h. We can construct a binary array $B = [b_1, b_2, \ldots, b_k]$ where $b_i \in \{0, 1\}$ for $1 \leq i \leq k$ and define $\sum_{i=1}^{k} b_i \cdot a_i = h$. That is, $b_i = 1$ if and only if a_i belongs to the sub-array of A whose sum equals h. By the construction of f in (1), this also means that tile T_i where $b_i = 1$ is colored. Notice that the number of colored cells in the one-dimensional Tilepaint instance equals $h = CR_1$, which implies that the one-dimensional Tilepaint instance has a solution.

Part 2: If the one-dimensional Tilepaint instance has a solution, then its corresponding subset-sum instance has a solution. Suppose we have a solution to the one-dimensional Tilepaint instance $(1, n, IG, CR, CC)$ where $CR = [CR_1]$ and CC is an array of length n containing -1. Suppose this instance contains p tiles and a solution exists by coloring some tiles (possibly none) among $T_1, T_2, \ldots, T_p$. Using f in (1), we define that if tile T_i where $1 \leq i \leq n$ is colored, then a_i, which corresponds to the size of T_i, is taken as an entry of a sub-array of A. Let us define a binary array $B = [b_1, b_2, \ldots, b_p]$ where $b_i \in \{0, 1\}$ and $b_i = 1$ if and only if T_i colored. Consequently, we also have $b_i = 1$ if and only if a_i is taken as an entry of a sub-array of A. The sum of all entries of such a sub-array is given by $\sum_{i=1}^{p} b_i \cdot a_i = CR_1 = h$. In other words, the corresponding subset-sum instance (A, h) has a solution. $\square$

The NP-completeness proof of the one-dimensional Tilepaint puzzles in Lemma 1 indirectly implies the NP-completeness of the general $m \times n$ puzzles if the tiles' shapes consist only of one row/column.

VIII. EXPERIMENTAL RESULTS

This section discusses the experimental result for measuring the actual running time of the proposed complete search and prune-and-search techniques. The experiment was conducted on a personal computer with 64-bit Windows 10 operating system and the algorithm was implemented using C++ programming language. The system used g++ compiler version 6.3.0 (MinGW.org GCC-6.3.0-1) with Intel(R) Core(TM) i7-8750H 2.20 GHz processor and 16 GB of RAM. We chose C++ because it tends to be faster than other well-known programming languages [36]. The repository regarding the verification algorithm, solver algorithms (both complete search and backtracking techniques), test cases (instance and solution of the puzzles), and the experimental results are provided at https://github.com/vincentiusar/Tilepaint-Solver.

The experiment considered the test cases collected from [37]. There are 250 test cases with three kinds of instance sizes, i.e., 10×10, 12×12, and 15×15. Each instance is guaranteed to have exactly one solution. The objective was to obtain the average execution time of three runs for the algorithms to solve an instance. Note that not all solutions to the instances can be found due to the limitation of our computational device.

From 250 test cases, only 48 test cases of size 10×10 can be solved using the complete search approach under 10 minutes. The minimum recorded running time is $4\,118$ ms, while the maximum is $554\,978$ ms. On average, the complete search method requires $149\,751.257$ ms to solve an instance of size 10×10 among 48 test cases.

Out of the 250 test cases, all test cases of size 10×10 and 12×12 are successfully solved using the prune-and-search algorithm in under 10 minutes. However, only 25 test cases of size 15×15 can be solved under the same time limit. Some 10×10 instances can be solved quickly in less than 0.001 ms.

From theoretical analysis, the asymptotic running time of the prune-and-search method is faster than the complete search approach by a factor of p. From the experimental results, the average running time of the complete search approach to solve a 10×10 instance is $149\,751.257$ ms, while the average running time of the prune-and-search method to solve instances of the same sizes is 311.954. This means the prune-and-search method is approximately 480 times faster than the complete search approach for solving 10×10 instance. We summarize the running times of the prune-and-search algorithm in Table I.

Size	Number of Test Case	Minimum Running Time	Maximum Running Time	Average Running Time
10×10	96	< 0.001	8 890.000	311.954
12×12	101	1.000	446 216.000	24 107.947
15×15	25	9.000	576 323.000	158 473.267

TABLE I: The running time (in milliseconds) for solving Tilepaint instance using the prune-and-search algorithm.

IX. CONCLUSION AND FUTURE WORKS

In this research, we present an algorithm to verify whether a Tilepaint configuration of size $m \times n$ containing p tiles satisfies the puzzle rules with a time complexity of $O(mn)$. We also propose two elementary search-based algorithms, the complete search and prune-and-search algorithms, for solving an arbitrary Tilepaint puzzle. In Theorem 4 and Theorem 5, we prove that the running time of the complete search and prune-and-search technique for solving an $m \times n$ Tilepaint instance with p tiles are respectively $O(2^p \cdot p \cdot mn)$ and $O(2^p \cdot mn)$, which implies that the latter method is asymptotically faster by a factor of p. We have also conducted an experiment to measure the performance of our proposed algorithms using test cases from [37]. The prune-and-search algorithm successfully solved all test cases of size 10×10 and 12×12, and some of the test cases of size 15×15. However, the complete search approach can only solve some test cases of size

10×10. Experimental results also show that the prune-and-search method are about 480 times faster than the complete-search method in solving some 10×10 instances.

We also address tractable and intractable variants of Tilepaint puzzles. The tractable variants of the Tilepaint puzzles resemble two-dimensional discrete tomography problems with complete information, where an $m \times n$ Tilepaint instance consists of mn tiles (i.e., all tiles are of size 1×1). These variants can be solved using a greedy algorithm with an asymptotic running time of $O(mn \log m + n \log n)$. The asymptotic upper bound for this algorithm can be made $O((mn)$ by using a linear time sorting algorithm, such as counting sort. We also prove that solving the general one-dimensional Tilepaint puzzle is NP-complete. Lemma 1 provides a polynomial-time reduction from the subset-sum problem to solving a one-dimensional Tilepaint. We prove that a subset-sum instance has a solution if and only if its corresponding one-dimensional Tilepaint instance has a solution.

Considering Tilepaint puzzles are NP-complete, we suggest exploring SAT-based solvers to solve the puzzle. The SAT-based technique is highly effective in solving another NP-complete puzzle such as Sudoku [18], [19], and we believe it could be applied to solving Tilepaint puzzles. Moreover, modifying the solver to find all possible solutions to a Tilepaint puzzle may provide some insights into the counting complexity of Tilepaint puzzles.

REFERENCES

[1] conceptispuzzles, "Cross-a-pix history," https://www.conceptispuzzles.com/index.aspx?uri=puzzle/cross-a-pix/history#:~:text=SingleClue%20Cross%2Da%2DPix%20are,by%20Toshiharu%20Yamamoto%20in%20Japan., Nov. 2022, accessed: 2022-11-21.

[2] W. Ting-Sheng, "Enhancing Problem-Solving Ability through a Puzzle-Type Logical Thinking Game," *Scientific Programming*, vol. 2022, pp. 1–9, 03 2022.

[3] 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.

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

[6] 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.

[7] C. Iwamoto and T. Ibusuki, "Computational Complexity of Two Pencil Puzzles: Kurotto and Juosan," in *Discrete and Computational Geometry, Graphs, and Games: 21st Japanese Conference, JCDCGGG 2018, Quezon City, Philippines, September 1-3, 2018, Revised Selected Papers 21*. Springer, 2021, pp. 175–185.

[8] R. Kaye, "Minesweeper is NP-complete," *Mathematical Intelligencer*, vol. 22, no. 2, pp. 9–15, 2000.

[9] C. Iwamoto and T. Ide, "Moon-or-Sun, Nagareru, and Nurimeizu are NP-complete," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 105, no. 9, pp. 1187–1194, 2022.

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

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

[12] 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.

[13] L. Robert, D. Miyahara, P. Lafourcade, L. Libralesso, and T. Mizuki, "Physical zero-knowledge proof and NP-completeness proof of Suguru puzzle," *Information and Computation*, vol. 285, p. 104858, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540121001905

[14] 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

[15] 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.

[16] 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.

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

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

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

[20] M. I. Putra, M. Arzaki, and G. S. Wulandari, “Solving Yin-Yang Puzzles Using Exhaustive Search and Prune-and-Search Algorithms,” *(IJCSAM) International Journal of Computing Science and Applied Mathematics*, vol. 8, no. 2, pp. 52–65, 2022.

[21] E. C. Reinhard, M. Arzaki, and G. S. Wulandari, “Solving Tatamibari Puzzle Using Exhaustive Search Approach,” *Indonesia Journal on Computing (Indo-JC)*, vol. 7, no. 3, pp. 53–80, Dec. 2022. [Online]. Available: https://socj.telkomuniversity.ac.id/ojs/index.php/indojc/article/view/675

[22] K. J. Batenburg and J. Sijbers, “DART: a practical reconstruction algorithm for discrete tomography,” *IEEE Transactions on Image Processing*, vol. 20, no. 9, pp. 2542–2553, 2011.

[23] S. Bilotta and S. Brocchi, “Discrete tomography reconstruction algorithms for images with a blocking component,” in *Discrete Geometry for Computer Imagery: 18th IAPR International Conference, DGCI 2014, Siena, Italy, September 10-12, 2014. Proceedings 18*. Springer, 2014, pp. 250–261.

[24] H. Ryser, “Combinatorial Properties of Matrices of Zeros and Ones,” *Canadian Journal of Mathematics*, vol. 9, pp. 371–377, 1957.

[25] G. T. Herman and A. Kuba, *Discrete tomography: Foundations, algorithms, and applications*. Springer Science & Business Media, 2012.

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

[27] J. Dreier, S. Ordyniak, and S. Szeider, “CSP Beyond Tractable Constraint Languages,” in *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[28] M. R. Garey and D. S. Johnson, *Computers and intractability*. W. H. Freeman San Francisco, 1979, vol. 174.

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

[30] B. Aspvall, M. F. Plass, and R. E. Tarjan, “A linear-time algorithm for testing the truth of certain quantified boolean formulas,” *Information processing letters*, vol. 8, no. 3, pp. 121–123, 1979.

[31] S. Brunetti and A. Daurat, “An algorithm reconstructing convex lattice sets,” *Theoretical Computer Science*, vol. 304, no. 1, pp. 35–57, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397503000501

[32] A. P. Stolk, “Discrete tomography for integer-valued functions,” Ph.D. dissertation, Leiden University, 2011.

[33] S. Asif, M. Coulombe, E. D. Demaine, M. L. Demaine, A. Hesterberg, J. Lynch, and M. Singhal, “Tetris is NP-hard even with $O(1)$ Rows or Columns,” *Journal of Information Processing*, vol. 28, pp. 942–958, 2020.

[34] B. Barak, “Introduction to theoretical computer science,” https://introtcs.org/public/lec_12_NP.html, 2023, accessed: 2023-3-27.

[35] K. Abrahamson, “The subset sum problem,” http://www.cs.ecu.edu/karl/6420/spr16/Notes/NPcomplete/ss.html, 2016, accessed: 2023-6-19.

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

[37] Otto Janko, “Tairupainto,” https://www.janko.at/Raetsel/Tairupeinto/index.htm, Oct. 2022, accessed: 2022-10-11.