A BACKTRACKING APPROACH FOR SOLVING PATH PUZZLES

Joshua Erlangga Sakti[1], Muhammad Arzaki[2*], Gia Septiana Wulandari[2]

[1] *Undergraduate Student, Computing Laboratory, School of Computing, Telkom University (40257)*
[2] *Computing Laboratory, School of Computing, Telkom University (40257)*
Email: [1]author1mail@mail.com, [2]author2mail@mail.com, [2]author3mail@mail.com
*Corresponding author

Abstract. We study algorithmic aspects of the *Path puzzle*—a logic puzzle created in 2013 and confirmed NP-complete in 2020. We propose a polynomial time algorithm for verifying an arbitrary path puzzle solution and a backtracking-based method for finding a solution to an arbitrary path puzzle instance. We prove that the asymptotic running time of our proposed method in solving an arbitrary Path puzzle instance of size $m \times n$ is $O(3^{mn})$. Despite this exponential upper bound, experimental results imply that a C++ implementation of our algorithm can quickly solve 6×6 Path puzzle instances in less than 30 milliseconds with an average of 3.02 milliseconds for 26 test cases.

Keywords: asymptotic running time, backtracking, NP-complete, Path puzzles

I INTRODUCTION

Path puzzle is a logic puzzle introduced by Roderick Kimball, a freelance puzzle maker, in his 2013 book [1] and was featured in The *New York Times*'s wordplay blog [2]. This puzzle is proven NP-complete by Bosboom et al. in 2020 [3]. It is played on a rectangular grid of cells with two openings on the edge and constraint numbers on some of the rows and columns.[1] The solution to this puzzle is a line connecting the two openings through the grid cells with each cell can only be passed once. The number of cells passing through a specific row or column must also equal to the constraint numbers on the rows or columns. There are some modifications to the original puzzle to increase the difficulty, such as using a non-rectangular grid and more than two openings.

Puzzles are a form of play that can improve mood and reduce stress from everyday life, thus mainly done as a form of recreational activity [4]. In addition, puzzles also can help develop some skills, such as logical problem-solving. Many puzzles are related to important computational and combinatorial problems, thus gathering the attention of scientific communities in computing and mathematics [5–8]. There have been many puzzles proven to be NP-complete, such as Corral Puzzle (2002) [9], Country Road (2012) [10], Hashiwokakero (2009) [11], Hiroimono (2007) [12], Juosan (2018) [13, 14], KPlumber (2004) [15], Kurotto [13, 14], Light Up (2005) [16], Minesweeper (2000) [17], Moon-or-Sun (2022) [18], Nagareru (2022) [18], Nonogram (1996) [19], Nurimeizu (2022) [18], Nurikabe (2003) [20, 21], Pearl Puzzle (2002) [22], Shikaku (2013) [23], Slither Link (2000) [24], Sudoku (2003) [25], Suguru (2022) [26], Tatamibari (2020) [27], Tilepaint (2022) [28], Yajilin (2012) [10], Yin-Yang (2021) [29], and ZHED (2022) [30].

[1] See `https://www.enigami.fun` for details.

The NP-completeness of the Path puzzle means a polynomial time verifier algorithm exists for verifying the Path puzzle's solution. This also means an exponential time solver exists to find the solution to arbitrary Path puzzles. Nevertheless, to the authors' knowledge, no further exploration regarding the algorithms to solve these puzzles has been discussed. There are various approaches to solving NP-complete puzzles, such as using an integer programming model [31, 32], SAT solver [33], and SMT solver [27]. Some studies also have been conducted regarding the elementary technique to solve puzzles, such as an exhaustive search approach for solving the Tatamibari puzzle [34] and the prune-and-search technique for solving Yin-Yang puzzle [35]. This paper focuses on the backtracking technique as a straightforward and elementary method of solving a Path puzzle, thus establishing the upper bound to the time complexity of finding a single solution to the instance. We demonstrate that the solution can be obtained in exponential time in terms of the puzzle's size.

We present our investigation of a backtracking approach for solving Path puzzles, divided into seven parts. Section II discusses the formal definition of a Path instance, Path configuration, Path solution, an array representation of a Path puzzle, and an overview of the NP-completeness of Path puzzles. Some findings on conditions where a Path puzzle has no solution are discussed in Section III. Section IV discusses an algorithm to verify whether a Path configuration is a Path solution in polynomial time. Our main backtracking-based algorithm for solving an arbitrary Path instance of size $m \times n$ in $O(3^{mn})$ time is discussed in Section V. Section VI discusses the experiments conducted related to the solver algorithm and its results. Lastly, Section VII gives the summary and conclusion of the paper as well as some potential future works.

II PRELIMINARIES AND PREVIOUS INVESTIGATIONS

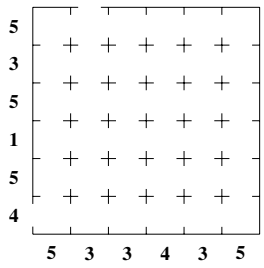
In this paper, we use one-based indexing for all arrays. Moreover, the i-th entry of a one-dimensional array A is denoted by $A[i]$, while the (i, j) entry of a two-dimensional array B is denoted by $B[i][j]$. This (i, j) entry refers to the component in row i and column j of B.

2.1 Formal Definition and Data Structure Representation of Path Puzzles

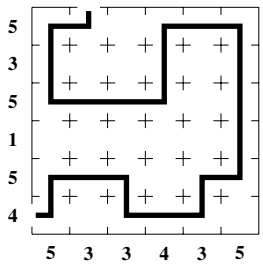
Before we discuss mathematical properties and algorithms related to the Path puzzle, we first discuss the formal definition of a Path puzzle instance, a Path puzzle configuration, and a Path puzzle solution.

Definition 1 *A Path puzzle instance (or a Path instance) of size $m \times n$ is a collection of mn cells represented as a grid of m rows and n columns where every cell is initially empty such that: (1) there are exactly two distinct cells (i_a, j_a) and (i_b, j_b) located at the edge of the grid indicating the doors (cells with opening) of the puzzle; and (2) there are constraint numbers for a subset of rows and columns, the constraint number for row i (if any) is denoted by cr_i and the constraint number for column j (if any) is denoted by cc_j.*

A Path puzzle is said to have a complete information if each row and column have a constraint number. A configuration for a Path puzzle instance (or a Path configuration) of size $m \times n$ is a grid of m rows and n columns with every cell is either empty or filled with exactly one horizontal, vertical, or L-shaped (turning) line. A Path puzzle solution (or a Path solution) is a configuration of a Path puzzle with exactly one non-intersecting path connecting the doors that passes through a number of cells in each row and column equal to the constraint number.



(a) An example of a Path puzzle instance.



(b) An example of Path puzzle solution for the instance in Figure 1a.

$$\begin{array}{cccccc} d & l & 0 & r & r & d \\ d & 0 & 0 & u & 0 & d \\ r & r & r & u & 0 & d \\ 0 & 0 & 0 & 0 & 0 & d \\ d & l & l & 0 & d & l \\ u & 0 & u & l & l & 0 \end{array}$$

(c) Array representation of the Path puzzle solution depicted in Figure 1b.

Figure 1. Examples of Path puzzle instance, solution, and array representation of a solution.

Examples of a Path puzzle instance and its corresponding solution is given in Figure 1. Notice that Figure 1b is obtained by adding a single non-intersecting continuous line between the doors in cells $(1, 2)$ and $(6, 1)$. In addition, the number of lines in each row and column satisfies the row and column constraints. For example, four lines occur in the sixth row, namely in cells $(6, 1)$, $(6, 3)$, $(6, 4)$, and $(6, 5)$.

To discuss the algorithmic approach to solving Path puzzles, we introduce data structures to represent Path puzzle instances, configurations, and solutions. An instance of a Path puzzle is represented using several variables, namely: (1) two positive integers m and n representing the size of the grid; (2) two pairs (i_a, j_a) and (i_b, j_b) representing the two cells with doors (located at the edge of the grid); and (3) two arrays cr of size m and cc of size n representing the rows and columns constraints, with the entry -1 signifying that particular row or column have no specified constraint number.

The entry $cr[i]$ denotes the constraint number for row i where $1 \leq i \leq m$. The value of $cr[i]$ is an integer between -1 and n (inclusive) with $cr[i] = -1$ if and only if the constraint number for row i in the Path puzzle instance is not defined. In this case, the number of lines occurring in row i is not specified. Similarly, the entry $cc[j]$ denotes the constraint number for column j where $1 \leq j \leq n$. The value of $cc[j]$ is an integer between -1 and m (inclusive) with $cc[j] = -1$ if and only if the constraint number for column j in the instance of the Path puzzle is unspecified. In this case, any number of lines can occur in column j. For example, to represent the Path puzzle instance in Figure 1a, we define $m = n = 6$, $(i_a, j_a) = (1, 2)$, $(i_b, j_b) = (6, 1)$, $cr = [5, 3, 5, 1, 5, 4]$, and $cc = [5, 3, 3, 4, 3, 5]$.

A configuration grid for a Path puzzle of size $m \times n$ is represented using a two-dimensional array of the same size whose entries are taken from the set $\{0, d, l, r, u\}$ where 0 denotes an empty cell and the characters d, l, r, and u respectively denote *down*, *left*, *right*, and *up* which are the direction of a line within a cell. For algorithmic purposes, in this paper, the cell corresponding to the destination door (i_b, j_b) in any Path puzzle configuration grid is always filled with the character u. Notice that this u can be changed with any character other than 0. Using this convention, the Path puzzle solution in Figure 1b is represented using the array in Figure 1c. Here, we add spaces between two characters in the same row for clarity.

2.2 Overview of the NP-Completeness of Path Puzzles

Path puzzles were recently proven to be NP-complete by Bosboom et al. in 2020 [3]. It is shown that Path puzzles are NP-complete even with complete information (i.e., every row and column has a constraint number). This NP-completeness infers that checking whether a configuration is also a Path puzzle solution to a particular Path puzzle instance can be done easily in polynomial time—but finding a solution to this instance currently needs an exponential number of steps in terms of the size of the puzzle. Moreover, any efficient (i.e., polynomial time) algorithm that solves general Path puzzles can be transformed into other algorithms for solving various NP-complete problems, thus solving the long-standing P versus NP problem.

Bosboom et al. stated that Path puzzles are closely related to 2D orthogonal discrete tomography, i.e., a problem to construct a black-and-white image given the number of black pixels in each row and column. An example of a discrete tomography problem is illustrated in Figure 2. Path puzzle is essentially a 2D discrete tomography problem with possibly partial information and Hamiltonicity constraint on the output. However, no further exploration of the algorithms to solve these puzzles have been discussed.

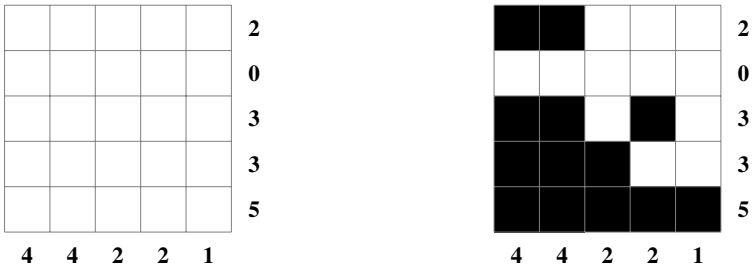


Figure 2. An example of a discrete tomography problem instance (left) and one of its solutions (right).

Bosboom et al. proved that the Path puzzle is NP-complete by a chain of parsimonious reductions from the source NP-complete problem of Positive-1-in-3-SAT—a problem to find an interpretation of a 3CNF formula ϕ with only positive literals where only one literal for every clause of ϕ is true. Parsimonious reduction means that the number of solutions is preserved between the source and the target instances. The reduction order is as follows. A modification to the reduction chain by Garey and Johnson [36] is used to reduce Positive 1-in-3-SAT problem to 3-Dimensional Matching [3, Theorem 2.2], 3-Dimensional Matching to Numerical 4-Dimensional Matching [3, Theorem 2.3], Numerical 4-Dimensional Matching to Numerical 3-Dimensional Matching [3, Theorem 2.4], Numerical 3-Dimensional Matching to Length Offsets Problem [3, Theorem 3.1], and finally Length Offset Problem to Path Puzzle Problem [3, Theorem 3.3].

Based on a classical result by Ryser [37] and a comprehensive investigation by Herman and Kuba [38], an arbitrary instance of discrete tomography with complete information is tractable, i.e., solvable in polynomial time. The instance can be verified to have a solution through a *majorization* technique and then solved with a greedy algorithm. This suggests that the factor that makes the Path puzzle NP-complete is the addition of the Hamiltonicity constraint. This is consistent with the statement from Bosboom et al. regarding the NP-hardness of Path puzzles with complete information.

Another puzzle that is similar to the Path puzzle is Nonogram, which was also proven to

be NP-Complete [19] [39]. Nonogram is played on a grid and requires the player to fill the cells according to the numerical clues on the side of the grid. A solved Nonogram usually reveals a hidden pattern. However, unlike in the Path puzzle—in Nonogram, there can be multiple numerical clues for a single row or column. For example, a 3 1 2 clue on a row means that there must be three consecutive filled cells, followed by one filled cell, and then two consecutive filled cells, where every segment of filled cells is separated by at least one blank cell. An example of a Nonogram puzzle is illustrated in Figure 3.

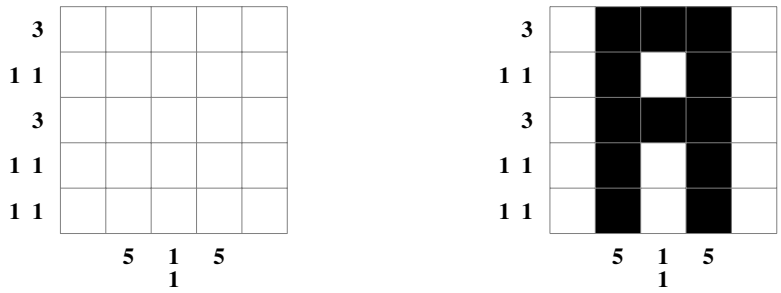


Figure 3. An example of a Nonogram instance (left) and its solution (right).

III CONDITIONS FOR THE NON-EXISTENCE OF PATH PUZZLE SOLUTION

In some conditions, a Path puzzle instance does not have any solution. We first discuss a condition for the non-existence of a solution to the Path puzzle with complete information. Recall that a Path puzzle instance has complete information if the constraint number for each row and column is defined. The following theorem is related to the 2D Discrete Tomography [37]. However, the proof of the pertinent theorem in [37] was omitted.

Theorem 1 *Let $sumR$ be the sum of all constraint numbers of all rows and let $sumC$ be the sum of all the constraint numbers of all columns in a Path puzzle instance with complete information. If the puzzle has at least one solution, then $sumR = sumC$.*

Proof. Suppose we consider a Path Puzzle instance with complete information and has at least one solution. According to Definition 1, when a path passes through a cell (i, j), it passes through a cell in row i and a cell in column j. Therefore, the total number of cells across all rows a path passes is equal to the total number of cells across all columns the path passes. We shall prove that $sumR = sumC$ using contradiction. Assume that $sumR \neq sumC$, then either $sumR > sumC$ or $sumR < sumC$. Notice that the condition $sumR > sumC$ implies that the path must pass one or more cells in one or more rows that are not a part of any column, which is impossible. The same reasoning also applies to the case if $sumR < sumC$. □

Notice that our proof of Theorem 1 uses the path property instead of the general property for 2D Discrete Tomography with complete information. In the subsequent theorem, we discuss the non-existence of a Path puzzle solution if the instance has one of the following criteria: (1) the doors are located in the same row and there is a row whose constraint number is 1, or (2) the doors are located in the same column and there is a column whose constraint number is 1. In the following theorem, the puzzle instance may have incomplete information.

Theorem 2 *Suppose we consider an $m \times n$ Path puzzle instance with $m, n > 1$. Let cr_i be the constraint number of row i and cc_j be the constraint number of column j. If the doors of this instance are cells (i_a, j_a) and (i_b, j_b) such that:*

1. *$i_a = i_b$ and there is a row p such that $cr_p = 1$, or*
2. *$j_a = j_b$ and there is a column q such that $cc_q = 1$,*

then the instance has no solution.

Proof. Suppose we consider a Path puzzle defined over a grid of size $m \times n$ with two doors (i_a, j_a) and (i_b, j_b). Firstly, let us assume that $i_a = i_b = r$. This condition also implies that $j_a, j_b \in \{1, n\}$ and $j_a \neq j_b$. A solution to this puzzle must begin at (r, j_a), traverse every row i where $cr_i > 0$, and end at (r, j_b). This implies that every row i with $cr_i > 0$ must be traversed twice. This is because the path must pass through a cell (i, c_a) in a row i where $1 \leq c_a \leq n$ and eventually traverse back to row r by passing through another cell (i, c_b) in a row i where $1 \leq c_b \leq n$ and $c_a \neq c_b$. Thus, any row i with $cr_i > 0$ is passed twice. With that in mind, if there is a row p such that $cr_p = 1$, then it is not possible to construct a non-intersecting path starting at (r, j_a) and ending at (r, j_b).

The same reasoning also applies if the doors are (i_a, c) and (i_b, c) and there is a column q where $cc_q = 1$. Here, $i_a, i_b \in \{1, m\}$ and $i_a \neq i_b$. If the doors are located in the same column $c \in \{1, n\}$, then any path connecting the doors must traverse every column j with $cc_j > 0$ at least twice. □

The following example illustrates Theorem 2 when the doors are located in the same column.

Example 1 Observe Figure 4. The two cells with a door are $(1, 1)$ and $(4, 1)$, which are in the same column and there is a column 2 with constraint 1, colored in red. To satisfy the constraint of columns 2, 3, and 4, the solution must pass through a cell in column 2 from column 1. Also, to reach the other cell with a door, the path must pass through a cell in column 2 again. Since the constraint of column 2 is only 1, the path cannot pass through any more cells in column 2 (filled with red block) to reach column 1, thus there is no solution.

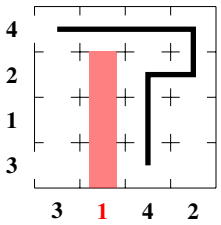


Figure 4. An Path instance where the two cells with a door are located in the same column and one of the column constraints is 1. The doors are located at $(1, 1)$ and $(4, 1)$.

IV VERIFYING PATH PUZZLES SOLUTIONS IN POLYNOMIAL TIME

Path puzzle is proven to be NP-complete by Bosboom et al. in 2020 [3], which means that verifying whether a Path puzzle configuration is also a Path puzzle solution can be done in polynomial time in terms of the size of the puzzle. In this section, we discuss algorithms for checking whether arbitrary Path puzzle configuration complies with the constraint and rules of a Path puzzle instance. A Path puzzle configuration of size $m \times n$ is represented as a two-dimensional array A with entries from the set $\{0, d, l, r, u\}$. $A[i][j] = 0$ indicates that cell (i, j) is empty. $A[i][j]$ being d, l, r, or u represents the direction *down*, *left*, *right*, and *up* in cell (i, j). The two cells with doors are represented as pairs (i_a, j_a) and (i_b, j_b). The constraint numbers of each row and column are correspondingly represented using one-dimensional arrays cr of size m and cc or size n. The value $cr[i]$ (resp. $cc[j]$) denotes the constraint number of row i (resp. column j). $cr[i]$ and $cc[j]$ are integers satisfying $-1 \leq cr[i] \leq n$ and $-1 \leq cc[j] \leq m$ for $1 \leq i \leq m$ and $1 \leq j \leq n$. If $cr[i] = -1$ (or $cc[j] = -1$), the constraint number for row i (or column j) is undefined.

The algorithm for verifying whether a Path puzzle configuration is also a Path puzzle solution is divided into two parts:

1. the algorithm for verifying the connectivity between two doors, that is, the algorithm to verify whether the cells (i_a, j_a) and (i_b, j_b) are connected by a single non-intersecting path, and
2. The algorithm for verifying compliance with row and column constraints by checking if $cr[i]$ (resp. $cc[j]$) equals the count of non-zero entries in the row i (resp. column j) of array A (provided that each of $cr[i]$ and $cc[j]$ are non-negative).

4.1 Verifying the Path Connectivity

Recall that a Path puzzle solution must be a single non-intersecting path connecting two cells with doors denoted by (i_a, j_a) and (i_b, j_b). Suppose A is a Path puzzle configuration of size $m \times n$. We first create a two-dimensional array $visited$ of the same size whose entries are initially set to 0. This array is created to track which cells have been visited by the path given in A. If a cell (i, j) is visited, the $visited[i][j]$ is set to 1. The idea of the path connectivity verification algorithm is as follows:

1. To check if the path in A connects two door cells (i_a, j_a) and (i_b, j_b), the algorithm verifies that $A[i_a][j_a]$ is non-zero and $A[i_b][j_b]$ is marked with u (as discussed in Section 2.1).
2. To check for non-intersecting paths in A, the algorithm traverses the path and marks visited cells in $visited$. If a cell (i, j) is marked as visited ($visited[i][j] = 1$) during the traversal, the path is considered intersecting and thus invalid.
3. To check that there is only one connected path in the solution, the algorithm compares the arrays A and $visited$ to check whether there is a cell (i, j) where $A[i][j] \neq 0$ and $visited[i][j] = 0$. If such a cell exists, then there is more than one path inside the grid and thus the configuration is not a valid solution.

The algorithm checks the path in configuration A by traversing it cell-by-cell, starting from (i_a, j_a). At each visited cell (i_c, j_c), the algorithm checks the direction specified in $A[i_c][j_c]$ and

moves to the next cell accordingly. For example, if $A[i_c][j_c] = d$, then the subsequent cell is (i_c+1, j_c) provided that the latter cell is within the $m \times n$ grid. To verify whether any subsequent cell during the traversal is within the $m \times n$ grid, we use the function IsValidCell(x, y) that returns true if and only if $1 \leq x \leq m$ and $1 \leq y \leq n$ holds. The traversal is performed until the path reaches the destination cell (i_b, j_b) or there is no more valid path. The steps are summarized in Algorithm 1.

Algorithm 1 IsConnected$(A, (i_a, j_a), (i_b, j_b))$ checks if a Path puzzle configuration A of size $m \times n$ satisfies the puzzle's connectivity rule for cells with doors (i_a, j_a) and (i_b, j_b).

Require: An $m \times n$ two dimensional array A denoting a Path configuration where $A[i][j] \in \{0, d, l, r, u\}$ and distinct pairs (i_a, j_a) and (i_b, j_b) where $1 \leq i_a, i_b \leq m$ and $1 \leq j_a, j_b \leq n$.
Ensure: The function returns true if the configuration A satisfies the puzzle's connectivity rule and false otherwise.

1: create a two dimensional array $visited$ with all entries set to 0
2: **if** $A[i_a][j_a] = 0$ **or** $A[i_b][j_b] \neq u$ **then**
3: **return false** ▷ the cells with doors are not connected by a path
4: **end if**
5: $(i_c, j_c) \leftarrow (i_a, j_a)$ ▷ (i_c, j_c) denotes the current visited cell in the traversal
6: **while** $(i_c, j_c) \neq (i_b, j_b)$ **and** $visited[i_c][j_c] \neq 1$ **do**
7: $visited[i_c][j_c] \leftarrow 1$
8: **if** $A[i_c][j_c] = l$ **and** IsValidCell$(i_c, j_c - 1)$ **then**
9: $(i_c, j_c) \leftarrow (i_c, j_c - 1)$ ▷ move to the left of the current cell
10: **else if** $A[i_c][j_c] = r$ **and** IsValidCell$(i_c, j_c + 1)$ **then**
11: $(i_c, j_c) \leftarrow (i_c, j_c + 1)$ ▷ move to the right of the current cell
12: **else if** $A[i_c][j_c] = u$ **and** IsValidCell$(i_c - 1, j_c)$ **then**
13: $(i_c, j_c) \leftarrow (i_c - 1, j_c)$ ▷ move to the top of the current cell
14: **else if** $A[i_c][j_c] = d$ **and** IsValidCell$(i_c + 1, j_c)$ **then**
15: $(i_c, j_c) \leftarrow (i_c + 1, j_c)$ ▷ move to the bottom of the current cell
16: **else**
17: **return false**
18: **end if**
19: **end while**
20: **if** $(i_c, j_c) \neq (i_b, j_b)$ **then**
21: **return false** ▷ the path does not end at (i_b, j_b)
22: **end if**
23: $visited[i_c][j_c] \leftarrow 1$
24: **for** $i \leftarrow 1$ to m **do**
25: **for** $j \leftarrow 1$ to n **do**
26: **if** $A[i][j] \neq 0$ **and** $visited[i][j] = 0$ **then**
27: **return false** ▷ there are more than one path in the grid
28: **end if**
29: **end for**
30: **end for**
31: **return true**

The time complexity of Algorithm 1 be determined from line 1 and the loops in lines 6-

19 and 24-30. Line 1 initializes an array of zeros of size $m \times n$, thus it runs in $O(mn)$. The maximum number of iterations for lines 6-19 equals the number of all cells in the grid. Thus the upper bound for the running time of these lines is $O(mn)$. The doubly nested loop in lines 24-30 runs m times for the outer loop and n times for the inner loop, and thus it runs in $O(mn)$. In conclusion, the upper bound for the running time of Algorithm 1 is $O(mn) + O(mn) + O(mn) = O(mn)$.

4.2 Verifying the Compliance of Rows and Columns Constraints

A Path puzzle solution must pass through a number of cells in each row and column, as specified by the constraint number. To verify this, the algorithm first creates copies of cr and cc and stores them respectively in crc and ccc. The algorithm then iterates through all entries in A and for every $A[i][j]$ that is non-zero, we decrement the value of $crc[i]$ and $ccc[j]$. Next, the algorithm checks the entries in arrays crc and ccc and if the number of cells traversed by the path matches the constraint number, all entries in both arrays should be 0, except for rows or columns without a constraint number in incomplete Path instances. We identify this by checking the values in cr and cc for those specific rows and columns. The steps of this algorithm is expounded in Algorithm 2.

Algorithm 2 COMPLYCONSTRAINT(A, cr, cc) checks whether a Path configuration of size $m \times n$ satisfies the constraint number of each row and column.

Require: A two dimensional array A of size $m \times n$ where each cell contains either d, l, r, u, or 0, array cr size m of integers between -1 and n (inclusive), and array cc of size n of integers between -1 and m (inclusive).

Ensure: The function returns true if the configuration A satisfies the constraint numbers in cr and cc; otherwise, the function returns false.

```
1:  initialize crc and ccc with the value from cr and cc, respectively
2:  for i ← 1 to m do
3:      for j ← 1 to n do
4:          if A[i][j] ≠ 0 then                          ▷ cell (i, j) is filled
5:              crc[i] ← crc[i] − 1
6:              ccc[j] ← ccc[j] − 1
7:          end if
8:      end for
9:  end for
10: for i ← 1 to m do
11:     if crc[i] ≠ 0 and cr[i] ≠ −1 then
12:         return false                  ▷ row i violates constraint number cr[i]
13:     end if
14: end for
15: for j ← 1 to n do
16:     if ccc[j] ≠ 0 and cc[j] ≠ −1 then
17:         return false                  ▷ column j violates constraint number cc[j]
18:     end if
19: end for
20: return true
```

The time complexity of Algorithm 2 can be analyzed as follows. Line 1 initializes two arrays crc and ccc respectively with the value cr and cc. These arrays are respectively of length m and n, and thus these initialization correspondingly take $O(m)$ and $O(n)$. The doubly-nested loop in lines 2-9 runs m times for the outer loop and n times for the inner loop. Hence the iteration in lines 2-9 takes $O(mn)$ time. The two loops in lines 10-15 and 16-21 take $O(m)$ and $O(n)$ times, respectively. As a result, assuming that $m, n \geq 1$, the upper bound for the running time of Algorithm 2 is $O(m) + O(n) + O(mn) + O(m) + O(n) = O(mn)$.

4.3 Main Verification Algorithm and Analysis

Suppose we are given a Path configuration represented in a two-dimensional array A. To check whether the Path configuration is also a Path solution, we use both Algorithms 1 and Algorithm 2 to verify all rules needed for a Path solution. If both functions return true, then the given Path configuration is a solution. Otherwise, the configuration is not a solution.

The time complexity of this verification algorithm can be determined from the time complexity of both Algorithm 1 and Algorithm 2. Since each of these algorithms runs in $O(mn)$ time, the overall running time complexity of the verification algorithm is $O(mn)$. In other words, we show that verifying whether a Path configuration is also a Path solution can be done in polynomial time in terms of the puzzle's size.

V SOLVING PATH PUZZLES USING BACKTRACKING APPROACH

This section discusses an algorithm for solving arbitrary Path puzzles of size $m \times n$. To find a solution, we generate some possible Path configurations and verify which configuration is also a solution. The function takes two pairs (i_a, j_a) and (i_b, j_b) which are the two door cells, and two arrays cr of size m and cc of size n which respectively store the constraint numbers of the rows and columns. The function also takes a pair (i_c, j_c), which represents the current cell position during the traversal. The function then returns a solution to the instance if it exists.

The solver initially creates a two-dimensional array A of size $m \times n$ for storing the grid state. All entries of array A are originally set to 0, indicating an empty grid. Before solving the puzzle, we check whether the instance complies with the conditions described in Theorem 1 and Theorem 2. If the instance complies with both conditions, the solver calls Algorithm 3 to start traversing and constructing the configuration from cell (i_a, j_a).

Algorithm 3 uses a backtracking method to construct the configuration, which is an optimization of the exhaustive search method. Backtracking adds pruning to candidates that do not lead to a solution. Therefore, the number of configurations that need to be verified decreases. Initially, the algorithm creates some variables to be used in the traversal as follows:

1. arrays $mx = [1, 0, 0, -1]$ and $my = [0, -1, 1, 0]$ to store the the possible movement for each cell (i, j), namely, $(i + 1, j)$, $(i, j - 1)$, $(i, j + 1)$, and $(i - 1, j)$;
2. an array $dir = [d, l, r, u]$ to store the possible directions for the path where $dir[i]$ corresponds to the pair $(mx[i], my[i])$, namely d corresponds to $(1, 0)$, l corresponds to $(0, -1)$, r corresponds to $(0, 1)$, and u corresponds to $(-1, 0)$;
3. a pair (i_{adj}, j_{adj}) to store the cell adjacent to cell (i, j).

In addition, the algorithm also calls some procedures and functions as follows:

1. ISVALIDCELL(i_{adj}, j_{adj}) is a function for checking whether cell (i_{adj}, j_{adj}) is inside the puzzle grid (i.e., $1 \leq i_{adj} \leq m$ and $1 \leq j_{adj} \leq n$).

2. DECREMENTENTRY(i_c, j_c) and INCREMENTENTRY(i_c, j_c) are respectively procedures to decrement and increment the value of $cr[i_c]$ and $cc[j_c]$ during the traversal if the current value is not -1, i.e., the row or column has a constraint number.

3. ISCONSTRAINTVALID(cr, cc) is a function to check whether every entry in cr and cc is either -1 or 0 after the traversal is completed. It uses the same principle as Algorithm 2 (particularly the steps in lines 10-19), namely, every entry that corresponds to a row or a column that has a constraint number should be 0 after the traversal.

Algorithm 3 TRAVERSECELL($A, (i_a, j_a), (i_b, j_b), (i_c, j_c), cr, cc$) traverses the cell (i_c, j_c) in a configuration array A, and then backtracks if necessary. The invocation TRAVERSECELL($A, (i_a, j_a), (i_b, j_b), (i_a, j_a), cr, cc$) starts the algorithm.

Require: A two-dimensional array A of size $m \times n$, each cell contains either d, l, r, u, or 0; pairs (i_a, j_a), (i_b, j_b), and (i_c, j_c) where $1 \leq i_a, i_b, i_c \leq m$, $1 \leq j_a, j_b, j_c \leq n$ ((i_a, j_a) and (i_b, j_b) are the doors); and arrays cr of size m and cc of size n respectively denoting the row and column constraints of the puzzles.

Ensure: The procedure outputs a solution to the puzzle if it exists or traverses to another possible cell.

1: $mx \leftarrow [1, 0, 0, -1]$; $my \leftarrow [0, -1, 1, 0]$; $dir \leftarrow [d, l, r, u]$
2: DECREMENTENTRY(i_c, j_c)
3: **if** $(i_c, j_c) = (i_b, j_b)$ **then**
4: **if** ISCONSTRAINTVALID(cr, cc) **then**
5: $A[i_c][j_c] \leftarrow u$ ▷ as defined in Section 2.1
6: output(A), terminate the algorithm ▷ A is a solution, algorithm terminates
7: **end if**
8: **else**
9: **for** $i \leftarrow 1$ **to** 4 **do**
10: $(i_{adj}, j_{adj}) \leftarrow (i_c + mx[i], j_c + my[j])$ ▷ generating four possible adjacent cells
11: **if** ISVALIDCELL(i_{adj}, j_{adj}) **and** $A[i_{adj}]][j_{adj}] = 0$ **and** $cr[i_{adj}], cc[j_{adj}] \neq 0$ **then**
12: $A[i_c][j_c] \leftarrow dir[i]$
13: TRAVERSECELL($A, (i_a, j_a), (i_b, j_b), (i_{adj}, j_{adj}), cr, cc$))
14: ▷ moves to cell (i_{adj}, j_{adj})
15: $A[i_c][j_c] \leftarrow 0$ ▷ backtracks from (i_{adj}, j_{adj})
16: **end if**
17: **end for**
18: **end if**
19: INCREMENTENTRY(i_c, j_c) ▷ backtracks from (i_c, j_c)
20: **if** $(i_c, j_c) = (i_a, j_a)$ **then**
21: output("no solution") ▷ backtracking from start cell (i_a, j_a) means no solution
22: **end if**

The algorithm first calls DECREMENTENTRY(i_c, j_c), indicating that the current cell is traversed. Next, the algorithm checks for the exit condition, which is when the current cell is the exit cell $((i_c, j_c) = (i_b, j_b))$. If the current cell is the second door cell, then the algorithm calls ISCONSTRAINTVALID(cr, cc) to check if the configuration is a solution. If this function returns true, then the algorithm marks the cell with u (as discussed in Section 2.1), outputs the solution array A, and it terminates.

If the current cell is not the second door cell, the algorithm begins the traversal to the next cell by first trying all possible orthogonally adjacent cells and then setting it to a pair (i_{adj}, j_{adj}). The algorithm checks whether the cell (i_{adj}, j_{adj}) is within the grid using the function ISVALIDCELL(i_{adj}, j_{adj}), checks whether $A[i_{adj}][j_{adj}] = 0$ to ensure that the traversal to the cell does not create an intersection, and checks whether both $cr[i_{adj}]$ and $cc[j_{adj}]$ are nonzero to ensure that the traversal complies with the numerical constraints. If all conditions are true, then the algorithm maps an entry in dir to $A[i_c][j_c]$ and assigns (i_{adj}, j_{adj}) to the current cell (i_c, j_c).

An example of traversal visualization is illustrated in Figure 5 in the form of a state space tree. Here we consider Path puzzle instance of size 3×3 whose doors are located at $(2, 1)$ and $(3, 2)$ with numerical constraints $cr = [3, 2, 2]$ and $cc = [2, 2, 3]$. All nodes in the tree represent the search space of the traversal, where each child node is the possible next state of the parent node. Each leaf node is marked with either a ✓ or a ✗ to indicate that the leaf is a valid or invalid solution, respectively. A solution is invalid because either the path cannot move to another adjacent cell or the path reaches the second door cell but does not comply with the numerical constraints.

If the traversal to (i_{adj}, j_{adj}) does not lead to a valid solution, the algorithm backtracks by resetting $A[i_{adj}][j_{adj}]$ back to 0 to try the following possible adjacent cell. If the same condition happens to every adjacent cell, then the algorithm backtracks from the current cell (i_c, j_c) by calling INCREMENTENTRY(i_c, j_c). When the algorithm gets to a point where it backtracks from the starting cell (i_a, j_a), then the algorithm outputs "no solution", signifying that the puzzle instance has no solution. The invocation TRAVERSECELL$(A, (i_a, j_a), (i_b, j_b), (i_a, j_a), cr, cc)$ is used to find a solution to a Path instance represented in a two-dimensional array A with two doors (i_a, j_a) and (i_b, j_b) with numerical constraints cr and cc. The asymptotic upper bound for the solver algorithm is discussed in Theorem 3.

Theorem 3 *The asymptotic upper bound for the running time of Algorithm 3 in solving an arbitrary instance of a Path puzzle of size $m \times n$ is $O(3^{mn})$.*

Proof. The algorithm traversal can be represented as a state space tree, where there are a maximum of three potential next states for every state in every level of the tree. This is because the three potential next states correspond to the possible directions of every cell, excluding the previously visited adjacent cell. This implies that there are at most 3^i states in level i of the tree. Since there are at most mn levels of the tree, i.e., the path visits all cells in the grid, the maximum number of states to be considered bounded by $1 + 3 + 3^2 + 3^3 + \cdots + 3^{mn} = (3^{mn+1} - 1)/2$, which is $O(3^{mn})$. □

VI COMPUTATIONAL EXPERIMENTS AND RESULTS

This section discusses the experiments conducted to test the running time of the proposed algorithm. The experiments were run on a 64-bit Windows 11 system using C++ programming

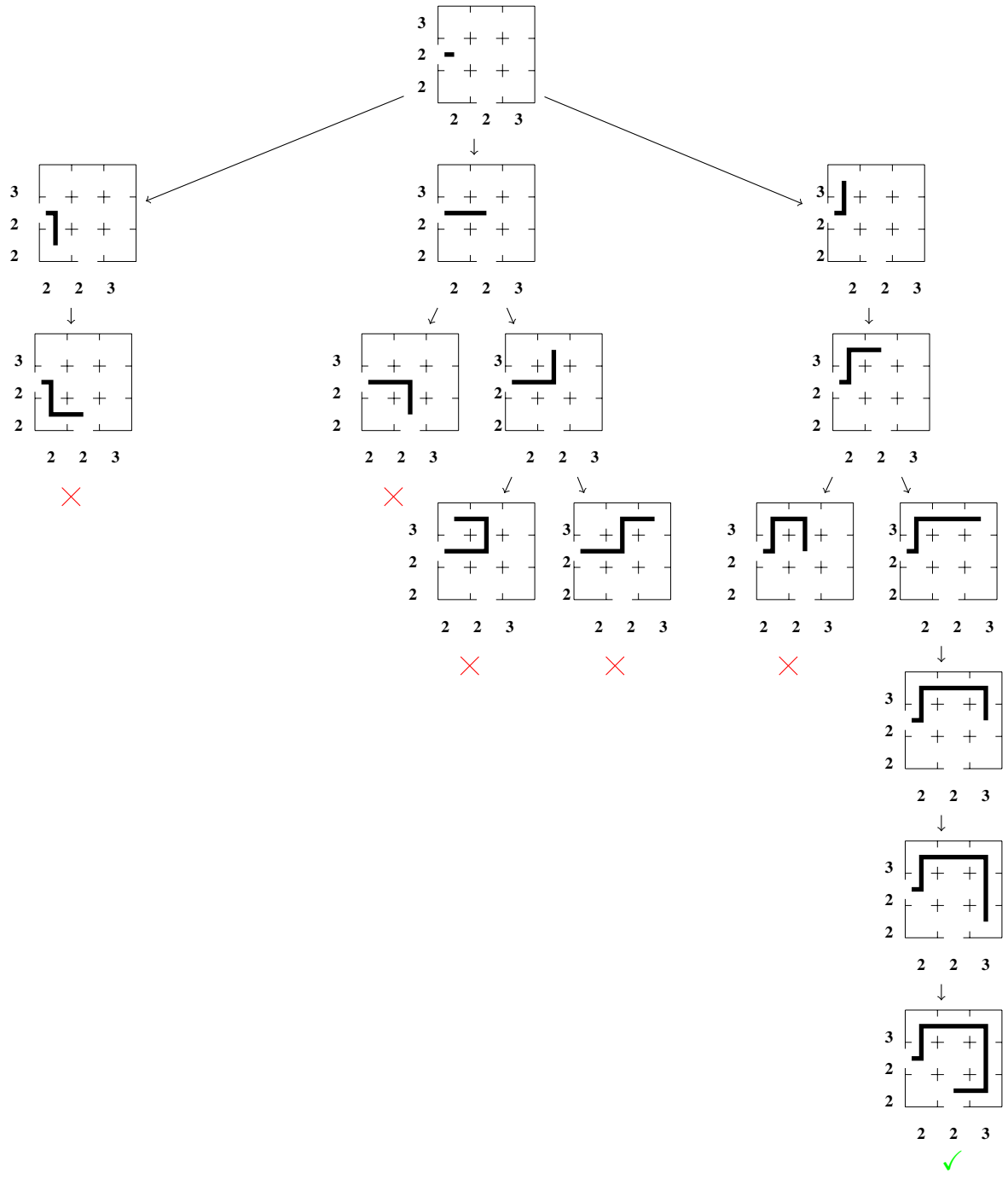


Figure 5. Visualization of the traversal in Algorithm 3 for a 3×3 Path instance. Here, the doors are located at $(2, 1)$ and $(3, 2)$. The numerical constraints are $cr = [3, 2, 2]$ and $cc = [2, 2, 3]$.

language. The reason for the usage of C++ is that it is relatively faster when compared to other popular programming languages [40]. The system uses g++ compiler version 6.3.0 and is equipped with AMD Ryzen 5 3550H @ 2.1GHz with 16GB of RAM. The source codes, test cases, and relevant documentation are provided for interested readers at `https://github.com/joshuagatizz/path-puzzles-backtracking-solver`.

In the experiment, Algorithm 3 was tested against 26 test cases where the solutions correspond to one of the standard English alphabets as described in [41]. Each test case considers a 6×6 grid with complete or incomplete information and only has one solution. The algorithm was run three times for each test case to determine its corresponding average running time. For some test cases, the implementation of our Algorithm successfully solved the puzzle quickly—in about 10^{-9} seconds.

The test cases with the fastest running time correspond to the puzzle representing the letter "B", "L", "P", "S", "U","V", "Y", and "Z" with an average running time of $\approx 10^{-9}$ s. On the other hand, the test case corresponding to the letter "M" is the slowest with an average running time of 27.34 milliseconds. The overall average running time for solving the 26 test cases is 3.02 milliseconds.

VII CONCLUSIONS AND FUTURE WORKS

We have discussed an algorithm to solve an arbitrary Path puzzle instance with complete or incomplete information using backtracking. In Theorem 3, we prove that the upper bound for the proposed algorithm to solve an instance of Path puzzle of size $m \times n$ is $O(3^{mn})$. We also conducted computational experiments to measure the running time of the proposed algorithm against 26 Path instances of size 6×6 as described in [41].

The experimental results imply that the algorithm is capable of quickly solving the test cases, with an average of 3.02 milliseconds and some test cases have a recorded running time on the order of 10^{-9} seconds. However, one should notice that the size of the puzzle instances in [41] is relatively small. Nonetheless, one interesting exploration is regarding a more efficient solver algorithm, possibly by using related properties to find the solution to a discrete tomography problem, such as the *majorization* technique from Ryser's paper [37]. In addition, since Path puzzles are NP-complete, we believe that the exploration and comparison between a backtracking solver and a SAT-based (or SMT-based) solver would be an interesting topic. Another interesting exploration is regarding an algorithm to find all solutions of a Path instance of size $m \times n$.

Some NP-complete puzzles have their non-trivial special cases that can be solved in polynomial time. For example, although the conventional Nonogram puzzle is NP-complete, a specific case in which every row or column contains only one block of connected cells can be converted to 2-SAT, and thus it belongs to the tractable class [42]. Another example is the Yin-Yang (Shiromaru-Kuromaru) puzzle. In general, this puzzle is NP-complete. However, the instance of Yin-Yang puzzles of sizes $m \times n$ where either m or n is less 3 is proven to be tractable [35]. We believe that finding a non-trivial special case of tractable Path puzzle variants would be an interesting future work.

REFERENCES

[1] R. Kimball, *Path Puzzles*. Enigami Puzzles & Games, 2013. [Online]. Available: https://books.google.co.id/books?id=tDhaswEACAAJ

[2] New York Times Wordplay, "Roderick kimball's path puzzles," https://archive.nytimes.com/wordplay.blogs.nytimes.com/2014/07/28/path-2/, Oct. 2022, accessed: 2022-11-14.

[3] 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

[4] S. Kim, "What is a puzzle," in *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*, 2008, pp. 35–39.

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

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

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

[8] R. Uehara, "Computational complexity of puzzles and related topics," *Interdisciplinary Information Sciences*, pp. 1–22, 2023.

[9] E. Friedman, "Corral puzzles are NP-complete," *Unpublished manuscript, August*, 2002. [Online]. Available: https://erich-friedman.github.io/papers/corral.pdf

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

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

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

[13] C. Iwamoto and T. Ibusuki, "Kurotto and Juosan are NP-complete," in *The 21st Japan Conference on Discrete and Computational Geometry, Graphs, and Games (JCDCG3 2018)*, 2018, pp. 46–48. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-90048-9_14

[14] C. Iwamoto and T. Ibusuki, "Polynomial-time reductions from 3SAT to Kurotto and Juosan puzzles," *IEICE Transactions on Information and Systems*, vol. 103, no. 3, pp. 500–505, 2020. [Online]. Available: https://www.jstage.jst.go.jp/article/transinf/E103.D/3/E103.D_2019FCP0004/_pdf

[15] D. Král, V. Majerech, J. Sgalla, T. Tichý, and G. Woegingerd, "It is tough to be a plumber," *Theoretical computer science*, vol. 313, pp. 473–484, 2004.

[16] B. McPhail, “Light Up is NP-complete,” *Unpublished manuscript*, 2005.

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

[18] C. Iwamoto and T. Ide, “Moon-or-Sun, Nagareru, and Nurimeizu are NP-complete,” *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, p. 2021DMP0006, 2022. [Online]. Available: https://www.jstage.jst.go.jp/article/transfun/advpub/0/advpub_2021DMP0006/_article/-char/ja/

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

[20] B. P. McPhail, “Complexity of puzzles: NP-Completeness results for Nurikabe and Minesweeper,” Ph.D. dissertation, Reed College, 2003.

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

[22] E. Friedman, “Pearl puzzles are NP-complete,” *Unpublished manuscript, August*, 2002. [Online]. Available: https://erich-friedman.github.io/papers/pearl.pdf

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

[24] Y. Takayuki, “On the NP-completeness of the Slither Link puzzle,” *IPSJ SIGNotes ALgorithms*, 2000.

[25] 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.

[26] 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

[27] 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

[28] C. Iwamoto and T. Ide, “Five Cells and Tilepaint are NP-Complete,” *IEICE Transcations on Information and Systems*, vol. 105, no. 3, pp. 508–516, 2022. [Online]. Available: https://www.jstage.jst.go.jp/article/transinf/E105.D/3/E105.D_2021FCP0001/_pdf

[29] 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.

[30] S. Saha and E. D. Demaine, "ZHED is NP-complete," in *Proceedings of the 34th Canadian Conference on Computational Geometry (CCCG 2022)*, 2022. [Online]. Available: https://erikdemaine.org/papers/Zhed_CCCG2022/paper.pdf

[31] 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.

[32] A. Bartlett, T. P. Chartier, A. N. Langville, and T. D. Rankin, "An integer programming model for the Sudoku problem," *Journal of Online Mathematics and its Applications*, vol. 8, no. 1, 2008.

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

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

[35] 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.

[36] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman & Co., 1979.

[37] H. Ryser, "Combinatorial properties of matrices of zeros and ones," *Canadian Journal of Mathematics*, vol. 9, pp. 371–377, 1957.

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

[39] H. J. Hoogeboom, W. A. Kosters, J. N. van Rijn, and J. K. Vis, "Acyclic constraint logic and games," *ICGA Journal*, vol. 37, no. 1, pp. 3–16, 2014.

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

[41] E. D. Demaine, "Path Puzzles Font," https://github.com/edemaine/font-pathpuzzles, Oct. 2022, accessed: 2022-10-11.

[42] 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