Solving Yin-Yang Puzzles Using Exhaustive Search and Prune-and-Search Algorithms

Made Indrayana Putra, Muhammad Arzaki, and Gia Septiana Wulandari

***Abstract*—We investigate some algorithmic and mathematical aspects of Yin-Yang/Shiromaru-Kuromaru puzzles. Specifically, we discuss two algorithms for solving arbitrary Yin-Yang puzzles, namely the exhaustive search approach and the prune-and-search technique. We show that both algorithms have an identical asymptotic running time of $O(\max\{mn, 2^{mn-h}\})$ for finding all solutions of a Yin-Yang instance with h hints of size $m \times n$. Nevertheless, our experiments show that the practical running time of the prune-and-search technique outperforms the conventional exhaustive search approach.**

***Index Terms*—complexity, exhaustive search, prune-and-search, Yin-Yang puzzles.**

I. Introduction

YIN-YANG (also known as Shiromaru-Kuromaru) is a pencil-and-paper puzzle first published in 1994 by the Japanese magazine Puzzler that has recently been proven to be NP-complete in [1]. In general, a Yin-Yang puzzle consists of $m \times n$ grid of cells, and every cell either has a black circle, a white circle, or is empty. The objective of this puzzle is to fill every empty cell with either a black or a white circle such that: 1) for each color (black and white), the cells containing circles of the same color form a single connected group of cells, where connectivity is based on four-way orthogonal adjacency; 2) there is no 2×2 grid of cells containing the same color.

Puzzles, particularly those based on pencil-and-paper, are primarily intended as recreational tools [2]. Nevertheless, the mathematical and computational aspects of puzzles have undergone significant investigations due to their connections to important combinatorial and computational problems. Several systematic studies have been carried out on the topic of the complexity of puzzles [3]–[5]. Moreover, many pencil-and-paper based puzzles have been proven to be NP-complete, such as (in chronological order): Sudoku (2003) [6], Nurikabe (2004) [7], Hiroimono (2007) [8], Heyawake (2007) [9], Hashiwokakero (2009) [10], Kurodoko (2012) [11], Yajilin and Country Road (2012) [12], Shikaku and Ripple Effect (2013) [13], Yosenabe (2014) [14], Shakashaka (2014) [15], Fillmat (2015) [16], Usowan (2018) [17], Sto-Stone (2018) [18], Dosun-Fuwari (2018), [19], Tatamibari (2020) [20], Kurotto and Juosan (2020) [21], and Yin-Yang (2021) [1].

Made Indrayana Putra is an undergraduate student at Computing Laboratory, School of Computing, Telkom University, Bandung 40257, Indonesia, email: themadeindrayana@gmail.com.

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

Manuscript received July 22, 2022; accepted September 2, 2022.

The NP-completeness of Yin-Yang puzzles means that there should exist a Yin-Yang solution verifier that can be executed in polynomial time. Moreover, there also should exist an exponential time algorithm for solving arbitrary Yin-Yang puzzles. Nevertheless, since algorithmic investigation relating to the Yin-Yang puzzle is relatively new and limited, to our knowledge, a formal investigation of the Yin-Yang solver has never been discussed rigorously. There are numerous approaches for solving NP-complete puzzles, such as using the integer programming model [15] or the SAT-solver technique [22]–[26]. Nevertheless, in this paper, we discuss two explicit yet elementary techniques for solving arbitrary Yin-Yang puzzles, namely the exhaustive search and prune-and-search approaches. We show that we can find all solutions of arbitrary Yin-Yang instances in exponential time in terms of the size of the puzzle and the number of hints.

The rest of the paper is organized as follows. We discuss some theoretical aspects of the Yin-Yang puzzles in Section II. Here, we also derive an additional rule regarding the non-existence of a 2×2 alternating pattern. In Section III we discuss an $O(mn)$ time algorithm for verifying whether an arbitrary $m \times n$ Yin-Yang configuration is also a solution. We discuss our main algorithms in Section IV and show that we can solve an arbitrary $m \times n$ Yin-Yang puzzles with h hints in $O(\max\{mn, 2^{mn-h}\})$ time. Section V discusses computational experiments of our algorithms. Here, we also discuss some combinatorial results based on mathematical analysis and experiments. Finally, this paper is summarized and concluded in Section VI.

II. Preliminaries

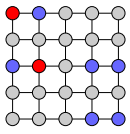
A. NP-Completeness of Yin-Yang Puzzles

The NP-completeness of Yin-Yang puzzles was first proven by Demaine et al. [1]. These puzzles can be considered as a type of *grid graph partitioning problem*. Here, we consider a rectangular $m \times n$ grid and the vertices are located in the intersection of the rows and columns (also called cells). Some vertices are pre-colored with either black or white (or any two distinct colors), and the objective of this problem is to color the remaining vertices subject to some constraints. The adjacency between vertices is defined as the usual four-way orthogonal adjacency in a grid, and an edge is defined as a connection between two vertices of the same color according to this adjacency rule. Mathematically, the *grid graph connected partition completion problem* and its related *grid graph tree partition completion problem* are defined below, and the problems were proven to be NP-complete [1].

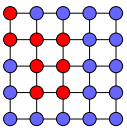
Definition 1. *[1] Suppose we consider an $m \times n$ grid and a graph $G = (V, E)$ is defined on this $m \times n$ grid. Suppose the vertices collection $\{A, B, U\}$ is a partition of V. The* grid graph connected partition completion problem *is a problem to determine whether there is a partition $\{A', B'\}$ of V such that $A \subseteq A'$, $B \subseteq B'$, and the induced subgraphs $G_{A'} = (A', E_{A'})$ and $G_{B'} = (B', E_{B'})$ are connected, i.e., $E_{A'} = \{\{u, v\} \mid \{u, v\} \in E \text{ and } u, v \in A'\}$ and $E_{B'} = \{\{u, v\} \mid \{u, v\} \in E \text{ and } u, v \in B'\}$. If the objective of the problem is restricted so that both $G_{A'}$ and $G_{B'}$ are trees, then the problem is called the* grid graph tree partition completion problem.

We illustrate the grid graph connected partition completion problem in Fig. 1 and the grid graph tree partition completion problem in Fig. 2. In Fig. 1a, we have a 5×5 grid graph and A, B, and U are respectively denote the set of red nodes, blue nodes, and gray nodes. The grid graph connected partition problem in Definition 1 examines whether there is another way to partition the nodes of the graph into two sets A' and B' where $A' \subseteq A$ and $B' \subseteq B$ and every node in the same sets are connected. In other words, we need to change every gray node in U into either a red node or a blue node and ensure that every node of the same color is connected according to the four-way orthogonal adjacency. A solution of the instance in Fig. 1a is given in Fig. 1b. However, since the solution is not a tree (we can see some red and blue cycles in the solution), it is not a solution for the grid graph tree partition completion problem. On the other hand, the solution shown in Fig. 2b for the same instance gives us a tree. Hence, it is a solution for the grid graph tree partition completion problem for the given instance.

In the grid graph connected partition problem, we have partition $\{A, B, U\}$ of V. To be compared with Yin-yang puzzles, the partition can be respectively considered as the set of cells with black circles, the set of cells with white circles, and the set of empty cells we have in the puzzle. This means that if we have induced connected subgraphs $G_{A'}$ and $G_{B'}$ of a grid graph, we can say that the first constraint of the Yin-Yang puzzle solution is satisfied. Moreover, if $G_{A'}$ and $G_{B'}$ are trees, the second constraint of the Yin-Yang puzzle solution is satisfied. It is proven that both grid graph connected partition completion and grid graph tree partition completion problems can be reduced to the planar 4-regular tree residue vertex breaking (TRVB) problem, which is an NP-hard problem described in [27].

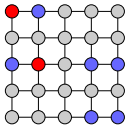


(a) An instance of a grid graph connected partition completion problem.

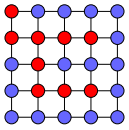


(b) A solution to the instance of a grid graph connected partition completion problem on the left.

Fig. 1: An illustration of the grid graph connected partition completion problem. The left part is the instance and the right part is the solution to the problem. Notice that there are four vertices resembling a 2×2 block of the same color in the solution.



(a) An instance of a grid graph tree partition completion problem.



(b) A solution to the instance of a grid graph tree partition completion problem on the left.

Fig. 2: An illustration of the grid graph tree partition completion problem. The left part is the instance and the right part is the solution to the problem. Notice that no four vertices resemble 2×2 block of the same color in the solution.

B. Yin-Yang Instance, Configuration, and Solution

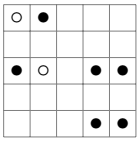
To construct algorithms for solving arbitrary Yin-Yang puzzles, we first provide the formal definitions of Yin-Yang instance, Yin-Yang configuration, and Yin-Yang solution in Definition 2 and Definition 3.

Definition 2. *An instance of a Yin-Yang puzzle (or Yin-Yang instance for short) of size $m \times n$ is a rectangular array (or board) of m rows and n columns such that:*

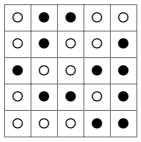
1) *a cell (i, j) is an intersection between row i and column j where $1 \leq i \leq m$ and $1 \leq j \leq n$,*
2) *every cell (i, j) is either empty or filled with either a white or a black circle.*

We call the initial white and black circles in a Yin-Yang puzzle as hints.

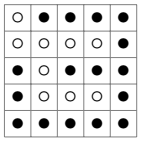
Definition 3. *An $m \times n$ Yin-Yang configuration is an $m \times n$ rectangular array whose each of the entries is either a black circle (which can be represented using 0) or a white circle (which can be represented using 1). A Yin-Yang solution is a Yin-Yang configuration that satisfies the two rules of the Yin-Yang puzzle.*



(a) A 5×5 Yin-Yang instance.



(b) A 5×5 Yin-Yang configuration.



(c) A solution for Yin-Yang puzzle in Fig. 3a

Fig. 3: Examples of a Yin-Yang instance, a Yin-Yang configuration, and a solution to a Yin-Yang puzzle. Fig. 3b is obtained from the instance in Fig. 3a by filling all cells with either colors. Fig. 3c is a solution to a Yin-Yang puzzle in Fig. 3a.

Fig. 3 provides an example of a Yin-Yang puzzle and its related terminology based on Definition 2 and Definition 3. Fig. 3a shows an example of a 5×5 Yin-Yang instance with eight circles as hints. This instance is used as a basis to solve the puzzle. Fig. 3b shows a 5×5 Yin-Yang configuration based on Fig. 3a. However, this configuration does not satisfy the connectivity rule for the Yin-Yang puzzle. Any Yin-Yang board whose every cell has been filled with either a black or a white circle is a Yin-Yang configuration. Fig. 3c is a solution for Yin-Yang instance in Fig. 3a. This solution is a Yin-Yang configuration that follows the two Yin-Yang rules. We notice that all Yin-Yang solutions are Yin-Yang configurations, but not conversely.

C. The Non-existence of 2×2 Alternating Cells

In a Yin-Yang puzzle, it is possible to have a configuration with 2×2 cells containing two white and two black circles that are placed diagonally. We formally define this condition in Definition 4 and illustrates such a condition in Fig. 4.

Definition 4. *A block of 2×2 alternating cells are four adjacent cells (r, c), $(r, c+1)$, $(r+1, c)$, and $(r+1, c+1)$ such that the color of the circles in (r, c) and $(r+1, c+1)$ are identical, the color of the circles in $(r, c+1)$ and $(r+1, c)$ are identical, and the color of the circles (r, c) and $(r, c+1)$ are different. In other words, all circles in the same row and the same column are of a different color.*

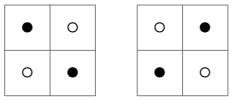
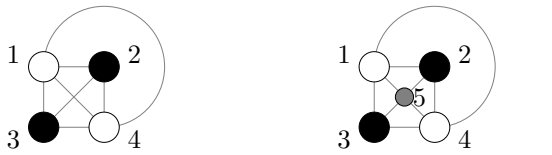


Fig. 4: Two types of 2×2 alternating patterns.

The non-existence of 2×2 alternating cells is discussed in [1, Lemma 1]. However, the proof is not described rigorously. Here, we discuss a more formal deduction to prove the non-existence of 2×2 alternating cells in any Yin-Yang solution by reducing the Yin-Yang configuration containing 2×2 alternating cells into a (non-planar) graph. Hence, the existence of 2×2 alternating cells implies the existence of a planar embedding of a previously known non-planar graph, namely, the complete graph K_5 [28]–[30]. Since it is already known that K_5 is non-planar, we need to know if we can convert a Yin-Yang configuration containing a 2×2 alternating pattern into a graph containing a subgraph that is isomorphic to K_5.

Theorem 1. *Let A be a Yin-Yang configuration of size $m \times n$ containing at least one 2×2 alternating cells, then this configuration does not satisfy the connectivity constraint of a Yin-Yang puzzle.*

Proof. Suppose we have 2×2 alternating cells in a configuration. We then have a subgraph with two white nodes and two black nodes in an alternating pattern. These nodes cannot be moved as they are reflecting the 2×2 alternating cells in the configuration. First, we assume that there is a path connecting the white nodes. If we add a new node with a different color in the middle of the subgraph, we will have a new subgraph with five nodes. Fig. 5 shows illustrations for both subgraphs. Now,



(a) A subgraph of a Yin-Yang configuration containing a 2×2 alternating pattern.

(b) A subgraph from Fig. 5a with an additional node in the middle.

Fig. 5: Graph construction from Yin-Yang configuration containing 2×2 alternating cells.

let us assume there is a path connecting the black nodes. Then we will have a connected graph containing five four-degree nodes. This means that the graph is isomorphic to K_5, which is known to be non-planar [28]–[30]. The new path cannot possibly be inside nodes 1, 2, 3, and 4, and must be outside of that area. If there is a path connecting the black nodes, the path must intersect the path of white nodes from node 1 to node 4. There is no way to connect the colored circles without intersecting with another color. Therefore, the existence of a 2×2 alternating pattern violates the connectivity rule. □

Another type of discrete proof using the extremal principle technique is discussed in [31]. As a result, we have the following corollary.

Corollary 1. *Any Yin-Yang solution does not contain 2×2 alternating cells defined in Definition 4.*

III. Polynomial Time Algorithm for Verifying Yin-Yang Solution

We discuss pertinent algorithms to verify whether a Yin-Yang configuration is a solution or not based on the Yin-Yang rules. The algorithms use 0-based indexing. Thus, an $m \times n$ Yin-Yang puzzle is represented with a two-dimensional array A and the indices for the rows and columns are respectively $0, 1, \ldots, m-1$ and $0, 1, \ldots, n-1$. We use a binary array A to represent a Yin-Yang configuration. The number 0 represents a black circle while 1 represents a white circle.

A. Constraint Checking Related to 2×2 Cells

We use Algorithm 1 to check whether a Yin-Yang board contains 2×2 cells of the same color or 2×2 cells of alternating color as described in Definition 4. In this algorithm, we call the procedure COMPARE2BY2(s) to check whether any 2×2 cells violate the Yin-Yang rule, i.e., whether these cells are of the same color or they form alternating patterns. More specifically, the procedure returns true if the 2×2 cells do not satisfy the aforementioned rule. This condition happens when s is the string 0000 (all circles are black), 1111 (all circles are white), 0110, or 1001. The 0110 and 1001 are strings associated with the alternating 2×2 cells as described in Definition 4.

The general idea of Algorithm 1 is to scan the binary two-dimensional array A of Yin-Yang configuration with a 2×2

Algorithm 1 CHECK2BY2($A[m,n]$) checks whether a Yin-Yang board A of size $m \times n$ contains a block of 2×2 cells of the same color or a block of 2×2 alternating cells as in Definition 4.

Input: A two dimensional binary array $A[m,n]$ of size $m \times n$, each cell contains either 0 or 1.
Output: The procedure returns true if $A[m,n]$ does not contain a block of 2×2 cells of the same color or a block of 2×2 alternating cells as in Definition 4; otherwise the procedure returns false.

```
1: for i ← 0 to m − 2 do
2:     for j ← 0 to n − 2 do
3:         s ← A[i][j] ‖ A[i + 1][j] ‖ A[i][j + 1] ‖ A[i + 1][j + 1]   ▷ concatenate all entries in a 2 × 2 adjacent cells
4:         if COMPARE2BY2(s) = true then
5:             return false
6:         end if
7:     end for
8: end for
9: return true
```

box and concatenate the value of the cells to form a string. The 2×2 box scans through every possible 2×2 cell in the array, and it is performed $(m-1)(n-1)$ times. The string is then compared with the COMPARE2BY2() procedure to check whether it satisfies the aforementioned rule. This process is illustrated in Fig. 6.

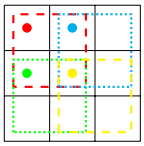


Fig. 6: Scanning process illustration for checking the rules related to 2×2 cells. The colored dots represent the top left-most cell in a 2×2 block in each iteration. The colored dashed boxes represent the 2×2 blocks checked in each iteration.

Fig. 6 shows a visualization of Algorithm 1. The colored dots represent the targeted cell in each iteration. The colored dashed squares represent the 2×2 blocks that are checked in each iteration. These 2×2 blocks are obtained by taking the targeted cell as the top-left cell in the block as well as three cells rightward, downward, and diagonally downward from it. Since the cells in row $m-1$ and column $n-1$ cannot form 2×2 cells where the cell is the top-left cell in the area, these cells are not included as the targeted cells.

The time complexity of this function can be observed from each for loop. The for loop in lines 1-8 runs $(m-1)$ times and the for loop in lines 2-7 runs $(n-1)$ times. Since the loops are nested, the algorithm runs in $O((m-1)(n-1))$ time, which can be simplified into $O(mn)$.

B. Connectivity Checking of a Yin-Yang Solution

In a Yin-Yang solution, all circles of the same color must be connected. To determine whether the circles are connected, we can use graph traversal algorithms, such as the Breadth-First Search (BFS) or the Depth-First Search (DFS) algorithm. Algorithm 2 is a function that checks whether the solution follows the connectivity constraint using the BFS approach. The main idea is to use BFS to search through each of the connected cells.

Before we use the BFS to check the connectivity, we initially determine the number of white and black circles in the board. We do this by checking each cell in the array A and counting the number of circles for each color. This procedure runs for all cells in the array A, which is of size $m \times n$. Thus, we have $O(mn)$ time complexity for this procedure.

We also need to know which cell is the starting point for the black circles and the white circles in our BFS procedure. Suppose we want to check whether the white circles are connected. First, we look for the leftmost top cell containing a white circle. Suppose this cell is denoted by (r,c). To check the connectivity of the white circles, we perform BFS using (r,c) as the initial node.

The algorithm uses arrays dr and dc to represent $(row, col+1)$, $(row+1, col)$, $(row, col-1)$, and $(row-1, col)$, which are the adjacent cells of the cell (row, col). The algorithm also calls the procedure VALIDCELL(r,c) to check whether a cell (r,c) cell exists within the array A (mathematically, $0 \leq r \leq m-1$ and $0 \leq c \leq n-1$). The algorithm then checks the following conditions:

1) If the adjacent cell has not been checked, then we mark its position, otherwise we ignore it.
2) If the value of that adjacent cell in the array A matches the color of the initial circle, then we add that position into the queue and add 1 to the count.

The algorithm then runs until there are no more cells to be checked inside the queue. The function returns the total number of the traversed cells with the same color as the initial cell. With this criterion, one can then see if it is equal to the total number of that color (0 or 1). If the condition holds, then all of those cells of the same color are connected. Otherwise, some cells of that color are not connected.

The running time of BFS is $O(|V|+|E|)$ where $|V|$ denotes the number of vertices, and $|E|$ denotes the number of edges in the graph [32, p. 597]. Since a cell is at most connected to four other cells, then $|E| \leq 4|V|$. We also have $|V| \leq mn$, thus the running time for Algorithm 2 is $O(|V|+|E|) = O(mn)$.

Another approach for checking the connectivity of the circles of the same color is using the DFS. In DFS, we use a stack data structure instead of a queue. Although theoretically, the asymptotic running time complexities of BFS and DFS

Algorithm 2 CHECKCONNECTIVITYBFS($A[m,n], r, c, color$) counts the number of connected cells of the same color in a Yin-Yang board A of size $m \times n$ using the BFS approach.

Input: A two dimensional binary array A of size $m \times n$, each cell contains either 0 or 1, two integers r and c that marks the starting position (a cell (r, c) within $A[m, n]$), and an integer $color$ whose value is either 0 or 1.
Output: The procedure returns the number of connected cells of a color based on $color \in \{0, 1\}$.

```
1:  checklist[r][c] ← 1                          ▷ adding an indicator that the first cell has been checked
2:  queue.push([r, c])
3:  dr ← [0, 1, 0, −1]
4:  dc ← [1, 0, −1, 0]
5:  count ← 1
6:  while queue not empty do
7:      row, col ← queue.pop()
8:      for i ← 0 to 3 do                                          ▷ iteration for dr and dc
9:          adjRow ← row + dr[i]
10:         adjRol ← col + dc[i]
11:         if ValidCell(adjRow, adjRol) and checklist[adjRow][adjCol] = 0 then
12:             checklist[adjRow][adjCol] ← 1        ▷ adding an indicator that the adjacent cell has been checked
13:             if A[adjRow][adjCol] = color then
14:                 queue.push([adjRow, adjCol])              ▷ push the row and column positions to the queue
15:                 count ← count + 1          ▷ increment the counter for the number of connected cells by 1
16:             end if
17:         end if
18:     end for
19: end while
20: return count
```

are identical, each algorithm might have a different practical running time. For experimental comparison purposes, we also devise an algorithm for checking the connectivity of a Yin-Yang solution using the DFS approach. The pseudo-code for this algorithm is similar to Algorithm 2 with little differences in lines 2, 6, 7, and 14 where the stack data structure is used instead of a queue.

C. Main Verification Algorithm and Its Analysis

Suppose we are given a Yin-Yang configuration represented in a two-dimensional array A. To check whether this configuration is also a Yin-Yang solution, we use Algorithm 3 which is a combination of Algorithm 1 and Algorithm 2. Notice that Algorithm 2 can be replaced with the DFS-based approach with the same asymptotic running time complexity. Some variables are written with the letter w or b in the beginning so that they can be associated with either white or black circles. For example, variables $wCount$ and $bCount$ respectively count the number of white and black circles in A, whereas variables $wStart$ and $bStart$ are correspondingly used to determine the starting positions of the white and black circles in A. All of these variables are later used for determining the connectivity of each color of circles using graph traversal algorithms. In Algorithm 3, the procedure CHECKCONNECTIVITYBFS() refers to either Algorithm 2 or its DFS-based counterpart. The $bValid$ variable counts the number of black circles traversed from the top leftmost black circle in A. Similarly, the $wValid$ variable counts the number of white circles traversed from the top leftmost white circle in A. If $bCount = bValid$, we conclude that all black circles in A are connected. Analogously, $wCount = wValid$ infers that all white circles in A are connected.

The asymptotic running time of Algorithm 3 can be determined from the functions involved in lines 1, 7, and 12. Observe that:

1) line 1 calls the procedure CHECKS2BY2() described in Algorithm 1 whose running time is $O(mn)$,
2) line 7 calls either the procedure CHECKCONNECTIVITYBFS() described in Algorithm 2 or its DFS-based counterpart, the running time of such an algorithm is $O(mn)$,
3) line 12 performs similar computation as line 7, the running time of such computation is $O(mn)$.

Since each procedure is called independently, the running time of Algorithm 3 is $O(mn)$. We conclude that verifying whether a Yin-Yang configuration of size $m \times n$ is also a Yin-Yang solution takes $O(mn)$ time. In other words, we show that verifying whether a Yin-Yang configuration is also a Yin-Yang solution takes polynomial time with respect to the size of the Yin-Yang board.

IV. SOLVING YIN-YANG PUZZLE USING EXHAUSTIVE SEARCH AND PRUNE-AND-SEARCH TECHNIQUES

In this section, we discuss two algorithms for solving arbitrary Yin-Yang puzzles with h hints of size $m \times n$. The algorithms also used 0-based indexing as described in Section III.

Algorithm 3 VERIFICATION($A[m, n]$) checks if an $m \times n$ Yin-Yang configuration is a valid solution or not.

Input: A two dimensional binary array A of size $m \times n$, each cell contains either 0 or 1.
Output: The procedure returns true if A is a valid Yin-Yang solution and false otherwise.

```
1: if CHECK2BY2(A) = true then
2:     bCount ← A.count(0)                          ▷ counts the number of black circles in A
3:     wCount ← A.count(1)                          ▷ counts the number of white circles in A
4:     bStart ← A.FIND_FIRST(0)                     ▷ finds the top leftmost cell filled with a black circle in A
5:     wStart ← A.FIND_FIRST(1)                     ▷ finds the top leftmost cell filled with a white circle in A
6:     if bStart ≠ (−1, −1) then                    ▷ if the top leftmost black circle exist
7:         bValid ← CHECKCONNECTIVITYBFS(A, bStart[0], bStart[1], 0)
8:     else
9:         bValid ← 0          ▷ since there is no black circle in A, the count for black circle is set to 0
10:    end if
11:    if wStart ≠ (−1, −1) then                    ▷ if the top leftmost white circle exist
12:        wValid ← CHECKCONNECTIVITYBFS(A, wStart[0], wStart[1], 1)
13:    else
14:        wValid ← 0          ▷ since there is no white circle A, the count for white circle is set to 0
15:    end if
16:    if bCount = bValid and wCount = wValid then
17:        return true
18:    end if
19: end if
20: return false
```

A. Exhaustive Search Technique for Finding Yin-Yang Solutions

An exhaustive search approach can be used to solve an arbitrary Yin-Yang puzzle. The idea is to find all possible configurations for a Yin-Yang puzzle and use the aforementioned Yin-Yang verification algorithm to find which configurations are also solutions. We first generate all possible Yin-Yang configurations based on an arbitrary Yin-Yang instance using Algorithm 4. A Yin-Yang instance is represented using an array whose entries are either 0, 1, or $*$, where 0 denotes a black circle, 1 denotes a white circle, and $*$ denotes an empty cell.

Algorithm 4 returns all possible configurations of a Yin-Yang instance. This procedure takes a copy of the input and checks every cell, whether it contains $*$, 1, or 0. If the current cell is filled with $*$ (which is associated with an empty cell), then the function takes all existing configurations and creates two new configurations for each of them. The new configurations have the current cell replaced with either 1 or 0, doubling the number of the previous configurations. This process repeats until all cells have been checked. The function then returns a list of all possible configurations, i.e., all binary arrays from the instance whose empty cells are filled with either 1 or 0.

Fig. 7 shows a simple decision tree visualization of each configuration generated with this algorithm from a single instance. By creating two new configurations from a single configuration, the procedure ends up with a larger number of unique configurations. In this example, the initial Yin-Yang instance consists of three empty cells, and each cell can be filled with either 0 or 1. As a consequence, there are eight different possible configurations as denoted by the leaves of the decision tree in Fig. 7.

Algorithm 4 GETCONFIGURATIONS($A[m, n]$) generates all $m \times n$ Yin-Yang configurations based on an $m \times n$ Yin-Yang instance.

Input: A two dimensional binary array A of size $m \times n$, each cell contains 0, 1, or $*$.
Output: The procedure returns all possible configurations from the given instance.

```
1:  config.push(A)
2:  for i ← 0 to m − 1 do
3:      for j ← 0 to n − 1 do
4:          if A[i][j] = ∗ then
5:              newList ← empty list
6:              while config not empty do
7:                  newConfig ← config.pop()
8:                  newConfig[i][j] ← 0
9:                  newList.push(newConfig)
10:                 newConfig[i][j] ← 1
11:                 newList.push(newConfig)
12:             end while
13:             config ← newList
14:         end if
15:     end for
16: end for
17: return config
```

The asymptotic time complexity of Algorithm 4 can be determined from the nested for loops in lines 2-16 and 3-15 and the while loop in lines 6-12. By considering the doubly-nested nature of the for loops in lines 2-16 and 3-15, we see that the algorithm performs at least $m \cdot n$ iterations to

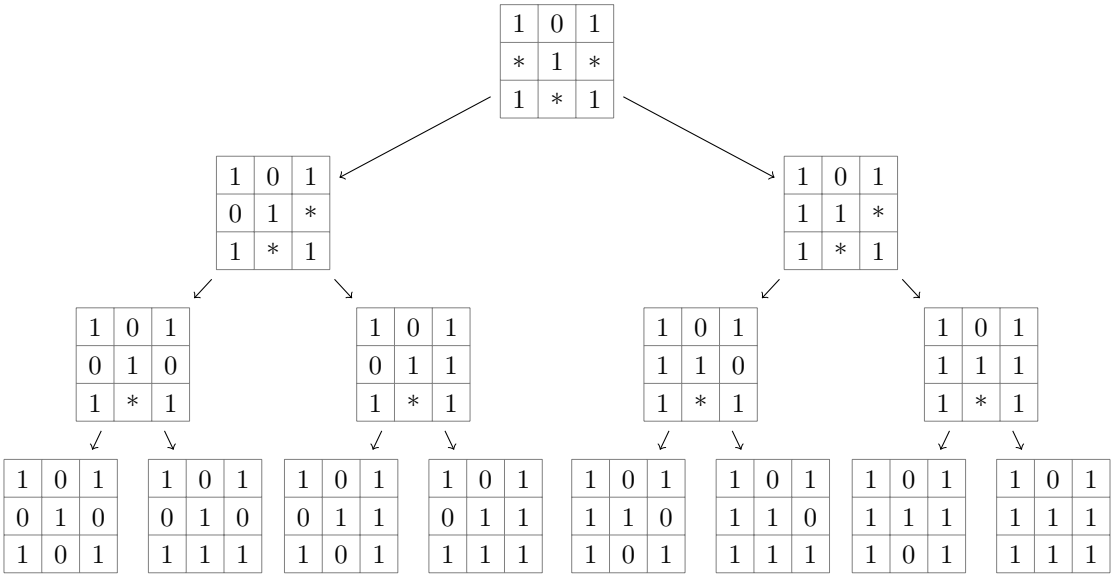


Fig. 7: Visualization of Algorithm 4. The root of the decision tree is the initial Yin-Yang instance where 0, 1, and $*$ correspondingly denote a black circle, a white circle, and an empty cell.

check whether a cell in the Yin-Yang configuration is empty (which is denoted by $*$). Notice that mn is the minimum number of iterations needed to perform a linear search in a two-dimensional array of size $m \times n$. The while loop in lines 6-12 runs depending on the number of *available unfinished configurations* in the variable $config$. This while loop empties the list $config$ and creates two new configurations. Each time the while loop terminates, the number of configurations doubles. Since the while loop runs if the current cell in the array A is equal to $*$, then the number of execution of this loop is $1+2+4+\cdots+2^{(mn-h)-1} = 2^{mn-h}-1$, where h is the number of hints (the number of non-empty cells). Notice that $mn-h$ is the number of empty cells in the Yin-Yang configuration. Furthermore, the value 2^{mn-h} also corresponds to the number of ways to fill every empty cell in the initial Yin-Yang instance with either 1 or 0. Moreover, the number of nodes in the decision tree illustrating the process of Algorithm 4 for an $m \times n$ Yin-Yang instance of h hints is $2^{mn-h+1}-1$. We conclude that the asymptotic time complexity of this while loop is $O(2^{mn-h})$.

Algorithm 4 takes precisely mn steps if the Yin-Yang instance does not contain an empty cell (a character $*$). Moreover, the algorithm takes precisely $2^{mn}-1$ steps if the Yin-Yang instance is empty (all cells are empty). The procedure requires $O(mn)$ time if the number of empty cells is sufficiently low, or mathematically we have $mn \geq 2^{mn-h}-1$. In contrast, if the number of empty cells is sufficiently high, we have $mn \leq 2^{mn-h}-1$, and thus the procedure requires $O(2^{mn-h})$ time. Therefore, we conclude that the asymptotic time complexity of Algorithm 4 is $O(\max\{mn, 2^{mn-h}\})$.

To obtain all solutions for an $m \times n$ Yin-Yang instance, we first generate all possible Yin-Yang configurations using Algorithm 4 and then verify each configuration using Algorithm 3. This process is described in Algorithm 5.

B. Prune-and-Search Technique for Finding Yin-Yang Solutions

We can optimize Algorithm 4 by applying the prune-and-search technique. N. Megiddo first suggested this paradigm for solving linear programming related problems [33]. In general, the prune-and-search approach removes (prunes) the non-promising search space in each iteration. Here, the idea is to check whether an unfinished configuration contains either 2×2 cells of the same color or a 2×2 alternating pattern. If any configuration contains any of these conditions, then such a configuration is removed from the prospective list of solutions.

Algorithm 6 provides a detailed description of the essential CHECKSURROUND2BY2() function in our pruning technique. Given a cell (r, c), this function checks whether (r, c) is located within a block of 2×2 cells of the same color or or a block of 2×2 alternating cells. To verify this condition efficiently, we consider four different positions of (r, c) within a block of 2×2 cells, namely top-left, top-right, bottom-left, and bottom-right. If (r, c) is at the top-left position within a block of 2×2 cells, then we also need to check the entries of $(r, c+1)$, $(r+1, c)$, and $(r+1, c+1)$. We then check whether these cells contain circles of the same color or constitute a 2×2 alternating pattern. Similar verification technique is performed if (r, c) is at the top-right, bottom-left, or bottom-right position of a 2×2 block.

Fig. 8 illustrates lines 3-12 of Algorithm 6. The dashed boxes illustrate the 2×2 cells that may contain the prohibited pattern. The red dot indicates the cell that is included in each of the possible 2×2 blocks. For example, the top-left dashed box demonstrates the condition when the cell (r, c) is located at the bottom-right position of the 2×2 block. Notice that the four possible positions of (r, c) within a 2×2 block are checked efficiently using a similar technique as in Algorithm 2. For instance, if $dr = dc = 1$, then Algorithm 6 checks whether the 2×2 block containing (r, c), $(r+1, c)$, $(r+$

Algorithm 5 FINDSOLUTIONEXHAUSTIVE($A[m,n]$) finds all solutions to a Yin-Yang instance of size $m \times n$ represented in an array A. Each cell of A contains either 0, 1, or $*$, where $*$ denotes an empty cell.

Input: An $m \times n$ Yin-Yang instance represented in a binary array $A[m,n]$.
Output: All solutions to the Yin-Yang instance $A[m,n]$ given as the input.

```
1: config ← GETCONFIGURATIONS(A[m,n])
2: solution ← empty list                    ▷ solution stores all possible solutions to the puzzle
3: for all C ∈ config do                    ▷ C is an array of Yin-Yang configuration
4:     if VERIFICATION(C) = true then
5:         solution.push(C)                 ▷ add C to the list of solutions
6:     end if
7: end for
8: return solution
```

Algorithm 6 CHECKSURROUND2BY2($A[m,n],r,c$) checks whether a cell (r,c) in a Yin-Yang board A of size $m \times n$ is contained within 2×2 cells of the same color or within 2×2 alternating cells as in Definition 4.

Input: A cell (r,c) within two dimensional binary array A of size $m \times n$.
Output: The function returns true if the cell (r,c) in A is not contained within 2×2 cells of the same color or within a 2×2 alternating cells as in Definition 4. The procedure returns false otherwise.

```
1: dr ← [1, −1, 1, −1]
2: dc ← [1, 1, −1, −1]
3: for i ← 0 to 3 do
4:     adjR = r + dr[i]
5:     adjC = c + dc[i]
6:     if VALIDCELL(adjRow, adjCol) then
7:         s ← A[r][c] || A[adjR][c] || A[r][adjC] || A[adjR][adjC]   ▷ concatenate all entries in a 2 × 2 adjacent cells
8:         if COMPARE2BY2(s) = true then
9:             return false
10:        end if
11:    end if
12: end for
13: return true
```

$1,c)$, and $(r+1,c+1)$ constitutes a prohibited pattern using COMPARE2BY2() function described earlier. Assuming that the addition and concatenation operations take constant time, it is obvious that the asymptotic complexity for the running time of Algorithm 6 is $O(1)$.

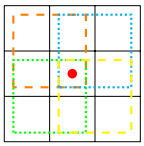


Fig. 8: Illustration for checking the rules related to 2×2 cells from a single cell.

Algorithm 7 is modified from the exhaustive search technique in Algorithm 4. The function performs a similar task as Algorithm 4 with a slight tweak when creating new configurations. Instead of creating two new configurations from an unfinished one, the function calls Algorithm 6 to check whether the new configuration contains prohibited 2×2 patterns. If the additional circle (a new value of either 0 or 1) results in any of these patterns, the new configuration is discarded.

Fig. 9 shows a visualization of Algorithm 7. This approach eliminates any configuration containing 2×2 block of the same color or any 2×2 alternating pattern. Thus, this approach may reduce the number of nodes in the decision tree significantly in some Yin-Yang instances. However, the asymptotic running time complexity for the worst-case scenario of Algorithm 7 is similar to that of Algorithm 4, i.e., $O(\max\{mn, 2^{mn-h}\})$. The resulting decision trees related to Algorithm 4 and Algorithm 7 are identical if we deal with an instance such that replacing $*$ by 0 or 1 never yields any type of prohibited 2×2 blocks.

To get all solutions to an $m \times n$ Yin-Yang instance, we first generate all possible Yin-Yang configurations that do not have prohibited 2×2 block using Algorithm 7. Afterwards, each of these possible configurations is then checked with Algorithm 3. Nevertheless, since the configurations do not contain 2×2 prohibited block, the CHECK2BY2() function in line 1 of Algorithm 3 can be skipped. This process is similar to that illustrated in Algorithm 5, but the function GETCONFIGURATIONS() is replaced with GETCONFIGURATIONSPRUNE().

V. COMPUTATIONAL EXPERIMENTS

Our experiments are mainly conducted to test the actual running time of our proposed algorithms. We run our experiments mainly using C++ programming language and g++ compiler version 11.2.0 in a Windows 10 64-bit operating system. We choose C++ instead of other programming languages such as

Algorithm 7 GETCONFIGURATIONSPRUNE($A[m, n]$) generates $m \times n$ Yin-Yang configurations based on an $m \times n$ Yin-Yang instance using prune-and-search technique.

Input: A two dimensional binary array A of size $m \times n$, each cell contains 0, 1, or $*$.
Output: The procedure returns some configurations that follows the 2×2 rule and do not have alternating pattern from the given instance.

```
config.push(A)
for i ← 0 to m − 1 do
    for j ← 0 to n − 1 do
        if A[i][j] = ∗ then
            newList ← empty list
            while config not empty do
                newConfig ← config.pop()
                newConfig[i][j] ← 0
                if CHECKSURROUND2BY2(newConfig, i, j) = true then
                    newList.push(newConfig)
                end if
                newConfig[i][j] ← 1
                if CHECKSURROUND2BY2(newConfig, i, j) = true then
                    newList.push(newConfig)
                end if
            end while
            config ← newList
        end if
    end for
end for
return config
```

Java or Python due to its relatively fast running time [34]. The system also uses Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz with 16.0 GB of RAM and a 500 GB SATA 2.5" SSD with up to 550 MB/s of reading speed and 520 MB/s of write speed. The source codes, input files, as well as other documents relating to our computational experiment are available at
`https://github.com/MadetheMeep/Yin-Yang-Exhaustive-Search`.

A. Experimental Results for Verification Algorithms

We tested our proposed verification function in Algorithm 3 with some Yin-Yang solutions as described in [35]. There are 26 different Yin-Yang solutions where each solution corresponds to a character in the standard English alphabet. Each solution is represented in an 11×9 board. For comparative purposes, we compare the utilization of the BFS and DFS approach in our proposed verification algorithm. In our experiment, we determine the average running time to verify every Yin-Yang solution. This average running time is obtained from three different runs for verifying each solution.

From our experiment, we obtain that the average running time for verifying a Yin-Yang solution using the BFS technique and DFS technique are respectively 0.5675 and 0.5711 milliseconds. The lowest average for verification time occurs on the letter "C" for BFS (0.4460 milliseconds) and the letter "J" for DFS (0.5010 milliseconds). In addition, the highest average running time happens on the letter "G" for BFS (0.7233 milliseconds) and the letter "X" for DFS (0.6307 milliseconds).

B. Experimental Comparison Between Exhaustive Search and Prune-and-Search Techniques

We tested our proposed algorithms for solving several modified Yin-Yang instances. These instances are modified from the Yin-Yang instances and solutions in [35]. For every instance that corresponds to an English alphabet, instead of using its original form in [35], we remove e circles from the original solution where e is varied between 0 and 23 (inclusive). The circles are removed in a left-to-right and top-to-bottom fashion except if the circles are also hints in the original Yin-Yang instance in [35]. For example, a Yin-Yang instance that corresponds to the letter "E" with $e = 16$ circles removed is obtained by removing 9 circles at the first row and 7 circles at the second row except those in sixth and seventh columns. The construction of such an instance guarantees that the solution of each modified Yin-Yang instance is unique.

We compare the running time of four different algorithms to solve Yin-Yang puzzles, namely the exhaustive search approach with BFS-based verification (ES-BFS), the exhaustive search approach with DFS-based verification (ES-DFS), the prune-and-search approach with BFS-based verification (PS-BFS), and the prune-and-search approach with DFS-based verification (PS-DFS). Each of these algorithms is tested to solve different Yin-Yang instances where each instance is obtained from the aforementioned modified instance in [35]. To obtain the average running time for solving an instance

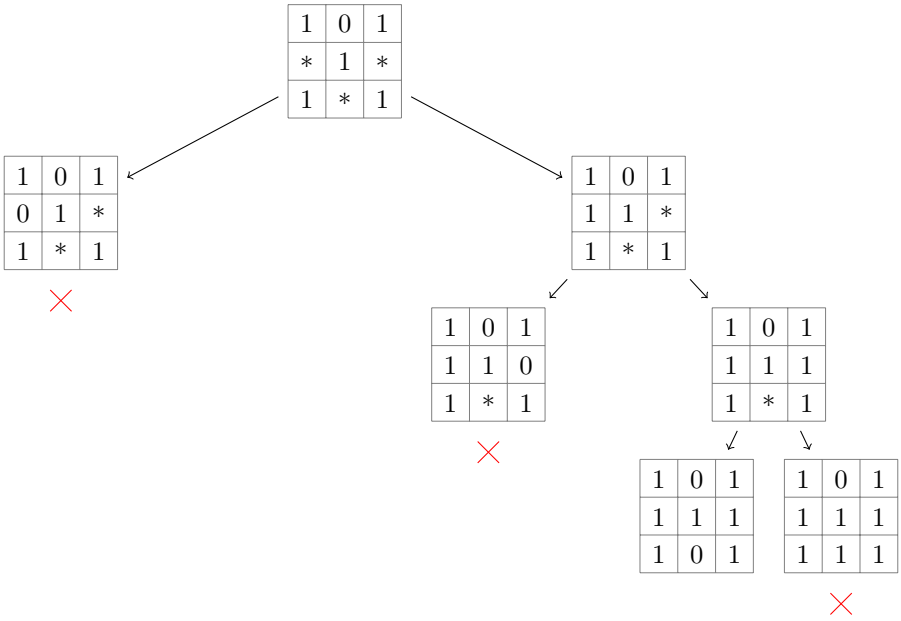


Fig. 9: Visualization of the prune-and-search technique. The instance is identical to that in Fig. 7. The red cross denotes that the updated instance contains a prohibited 2×2 block.

with e removed circles, the average running time of three runs for solving each of 26 such instances is computed. We summarize the relationship between the value of e and the practical running time of each algorithm in Fig. 10.

Notice that the number of removed circles is equal to the number of non-empty cells, thus $e = mn - h$, where h is the number of hints in the Yin-Yang instance. From Fig. 10 we see that the practical running time for solving the Yin-Yang instance of e removed circles grows exponentially in terms of e. This is consistent with the previously discussed theory in Section IV stating that the asymptotic running time for solving a Yin-Yang instance of size $m \times n$ with h hints is $O(\max\{mn, 2^e\})$ where $e = mn - h$. In addition, we see that despite both the exhaustive search and prune-and-search techniques having identical asymptotic running time complexities in the worst-case scenario for solving a Yin-Yang instance of size $m \times n$ with h hints, the prune-and-search approach practically outperforms the exhaustive search techniques. Moreover, our experiment also shows that finding a solution with the DFS-based verification is typically faster than obtaining a solution with the BFS-based verification, albeit their difference is not significant.

C. Counting the Number of Solutions of Yin-Yang Puzzles

In this section, we describe algorithms for finding the number of solutions to the Yin-Yang puzzle. Specifically, we remove all hints from the puzzles which are used to find solutions and instead start from an empty grid. We count the number of ways to fill this grid with valid Yin-Yang solutions. For a grid of size $m \times n$, we define $S(m, n)$ as the number of valid Yin-Yang solutions for an empty $m \times n$ grid. Since an empty Yin-Yang instance of size $m \times n$ has an identical number of solutions to an empty Yin-Yang instance of size $n \times m$, our investigation only considers the case when $m \leq n$. We also count the number of solutions for several empty grids using combinatorial techniques. These numbers are then compared to the output we obtain from the implementation of our algorithms.

We first derive mathematical facts related to the number of solutions for empty Yin-Yang instances of size $1 \times n$ and $2 \times n$.

Theorem 2. *There are $2n$ different valid Yin-Yang solutions to a $1 \times n$ empty Yin-Yang board where n is a positive integer.*

Proof. Suppose the number of white circles is w and the number of black circles is b. Since every cell is filled, we have $0 \leq w, b \leq n$ and $w + b = n$. There are two types of possible configurations that include both colors. Either all white circles are located on the left part of the board, or all white circles are located on the right part of the board. It is impossible to have white circles in between the black circles (or vice versa) since this configuration violates the connectivity rule (see Fig. 11 for an illustration). There is exactly one solution with w white circles, where $0 < w < n$, if all white circles are located on the left part of the board. Thus, the number of solutions with at least one white circle located on the left part of the board is $n - 1$. The same argument can be applied when all white circles are located on the right part of the board. Consequently, there are $2(n - 1)$ different solutions with at least one circle of each color. Notice that the board can be filled with only one color (all white or all black circles) and still results in a valid board. Hence, the total number of possible solutions is $2(n - 1) + 2 = 2n$. □

There are exactly 12 different solutions to an empty 2×2 Yin-Yang board. This quantity can be obtained using the following reasoning. Notice that there are $2^4 = 16$ possible ways to fill a 2×2 board with either a white or a black circle

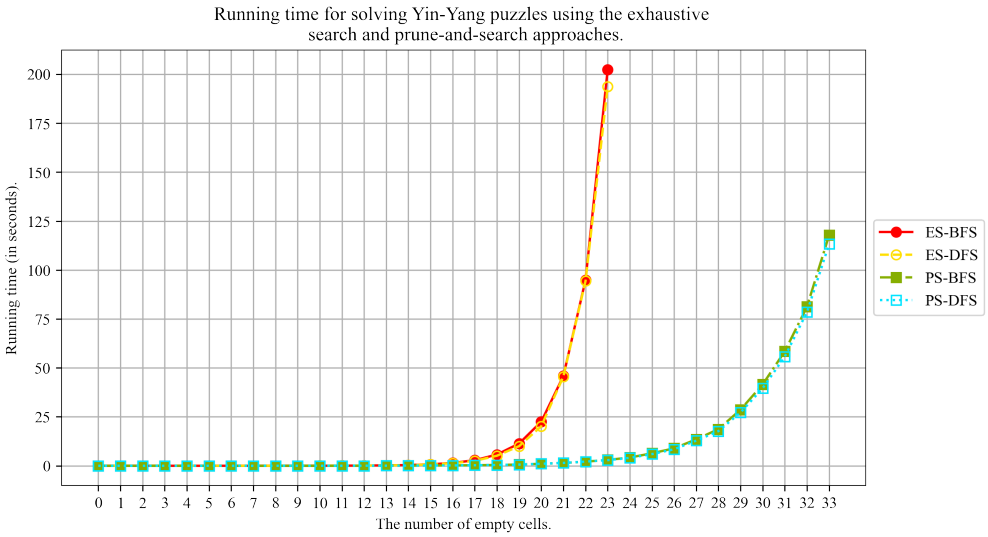


Fig. 10: The average running time for solving modified Yin-Yang instances with e removed circles from the original Yin-Yang solutions. Here, e satisfies $0 \leq e \leq 23$. Moreover, for the prune-and-search technique, the experiment is conducted until $e = 33$.

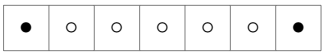


Fig. 11: An example of white circles in between black circles.

in each of the cells. However, since there are four types of prohibited 2×2 blocks, the number of valid Yin-Yang solutions of size 2×2 is $16-4=12$. The following lemma and theorem discuss the number of solutions to an empty $2 \times n$ Yin-Yang puzzle if $n \geq 3$.

Lemma 1. *The minimum (respectively, maximum) number of white (or black) circles in a Yin-Yang solution of a* $2 \times n$ *Yin-Yang puzzle where* $n \geq 3$ *is* $n-2$ *(respectively,* $n+2$*).*

Proof. Suppose the number of white and black circles are w and b, respectively. Since all cells are filled, then we have $w+b=2n$. Suppose that the number of white circles is less than $n-2$. Without loss of generality, we may assume that $w = n-3$ and infer that $b = n+3$. If the configuration contains a 2×2 block of the same color, then we are done (such a condition violates the 2×2 rule). Thus, we now assume that the configuration does not contain any 2×2 block of the same color. Since there are only two rows on the Yin-Yang board, then by the pigeonhole principle, one row is filled with at least three black circles. Now, consider the case if one row is filled with exactly three black circles and another row is filled with all black circles (see Fig. 12 for illustration). Because

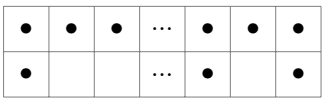


Fig. 12: A Yin-Yang board with exactly three black circles in the bottom row.

there is no 2×2 block of the same color, then a black circle is located between two white circles. This breaks the connectivity rule since the white circles are separated by a black circle in between (see Fig. 13). Hence, there is no solution for any $2\times n$ Yin-Yang configuration with $w = n-3$ and $b = n+3$ and vice versa. With the same reasoning, there is no solution for any $2 \times n$ Yin-Yang configuration with $w \leq n-3$ and $b \geq n+3$ and vice versa. Thus, the minimum and the maximum number of black or white circles in a $2 \times n$ Yin-Yang puzzle are respectively $n-2$ and $n+2$. □

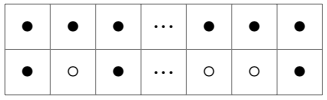


Fig. 13: A Yin-Yang configurations with a black circle in between white circles.

We use the previous lemma to prove the following theorem.

Theorem 3. *There are exactly* 18 *different valid Yin-Yang solutions to a* $2 \times n$ *empty Yin-Yang board where* $n \geq 3$*.*

Proof. Suppose the number of white circles and black circles are respectively denoted by w and b. Since all the cells are filled, we have $w+b=2n$. According to Lemma 1, we have $n-2 \leq w, b \leq n+2$. With this fact, five separate quantities can be defined as follows:

1) W_{n-2}: the number of solutions to a Yin-Yang puzzle with $n-2$ white circles (and $n+2$ black circles),
2) W_{n-1}: the number of solutions to a Yin-Yang puzzle with $n-1$ white circles (and $n+1$ black circles),
3) W_n: the number of solutions to a Yin-Yang puzzle with n white circles (and n black circles),
4) W_{n+1}: the number of solutions to a Yin-Yang puzzle with $n+1$ white circles (and $n-1$ black circles),

5) W_{n+2}: the number of solutions to a Yin-Yang puzzle with $n+2$ white circles (and $n-2$ black circles),

First, we determine W_{n-2}. Here, we have $n-2$ white circles and $n+2$ black circles. Therefore, to put the black circles, a row must be filled with all black circles, and the first and last columns of the other row must be filled with black circles. An illustration of this condition can be seen in Fig. 14. Notice

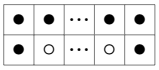


Fig. 14: A possible Yin-Yang solution for W_{n-2}.

that the configuration can be mirrored vertically and creates another configuration where the white circles are in the first row. Thus, we have $W_{n-2} = 2$.

Second, we determine W_{n-1}. In this case, we have $n-1$ white circles and $n+1$ black circles. To put the black circles, a row must be filled with all black circles and one of the cells in another row must be filled with one black circle, which can be the first or the last column. One such condition is illustrated in Fig. 15. Notice that the configuration can be mirrored both

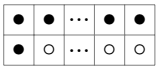


Fig. 15: A possible Yin-Yang solution for W_{n-1}.

horizontally and vertically. This creates another solution where there are two black circles in the last column, all the white circles in the first row, or both. Thus, we have $W_{n-1} = 4$.

Third, we compute W_n where there are n black and white circles each. There are two methods to fill the Yin-Yang board. Either by filling one row with all white and another row with all black or by filling the first column with all white and the last column with all black. Two examples of such conditions are illustrated in Fig. 16. Notice that in Fig. 16, the solution

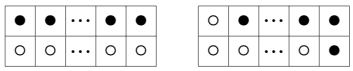


Fig. 16: Two possible Yin-Yang solutions for W_n.

on the left can be mirrored vertically, while the solution on the right can be mirrored both horizontally and vertically. Thus, we have $W_n = 2 + 4 = 6$.

We can obtain the result of W_{n+1} and W_{n+2} by observing W_{n-1} and W_{n-2}. Notice that the colors are swapped from those in the argument of W_{n-1} and W_{n-2}, and thus the number of possible solutions are identical. Hence, we have $W_{n+1} = W_{n-1}$ and $W_{n+2} = W_{n-2}$. Consequently, the number of possible solutions for an empty $2 \times n$ Yin-Yang puzzle where $n \geq 3$ is $W_{n-2}+W_{n-1}+W_n+W_{n+1}+W_{n+2} = 2(2) + 2(4) + 6 = 18$. □

We also investigate the number of solutions for Yin-Yang puzzles of size $3 \times n$, $4 \times n$, and $5 \times n$. However, this investigation is exclusively conducted using computational experiments as we have not found any formula for counting the number of solutions for such Yin-Yang puzzles. We represent the result of our investigation for several values of $S(m, n)$ in Table I. However, one should note that some solutions are identical to one another if we consider horizontal or vertical reflection, or 90° or 180° rotation in the clockwise or counterclockwise direction.

TABLE I

The value of $S(m, n)$ for several m and n where $1 \leq m \leq n \leq 6$.

m \ n	1	2	3	4	5	6
1	2	4	6	8	10	12
2		12	18	18	18	18
3			34	50	70	94
4				96	220	420
5					660	1948

Our computational experiments are also used to determine the actual running time for counting the number of solutions to the empty Yin-Yang puzzles of several sizes. We compare the actual execution times of four different algorithms as described in Section V-B, namely ES-BFS, ES-DFS, PS-BFS, and PS-DFS. Fig. 17 shows the average running times of three runs for finding the number of solutions to empty instances of $1 \times n$ Yin-Yang puzzles, where $1 \leq n \leq 25$ for each algorithm. The actual running time of each four different approaches is relatively similar to one another for each n. Here, the prune-and-search approach does not outperform the exhaustive search technique. This happens because the pruning algorithm only checks if the configuration contains any type of prohibited 2×2 block. Since the size of the Yin-Yang puzzle is $1 \times n$, there are no 2×2 prohibited blocks in the configuration. Thus, no configurations are eliminated using the prune-and-search approach, and the verification algorithm needs to evaluate every possible configuration.

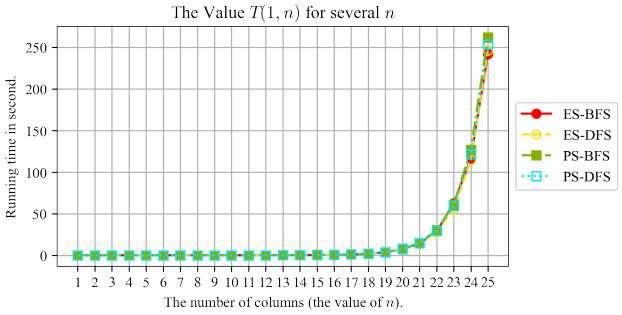


Fig. 17: The average running time for finding the number of solutions to empty instances of $1 \times n$ Yin-Yang puzzles where $n \in [1, 25]$.

Fig. 18 shows the average running time for finding the number of solutions to empty instances of $2 \times n$ Yin-Yang puzzles where $2 \leq n \leq 12$. Here, we notice some differences between the average running time of the regular exhaustive search approach and its prune-and-search counterpart for $n \geq 9$. The higher the value of n, the bigger the differences between the actual average running time of these two algorithms. This happens because the prune-and-search algorithm reduces

the number of configurations that need to be evaluated by removing the prohibited 2×2 blocks. The bigger the Yin-Yang board dimension, the more it reduces the number of configurations. In addition, the experimental result also shows that there is no significant difference between the BFS-based and DFS-based verifications.

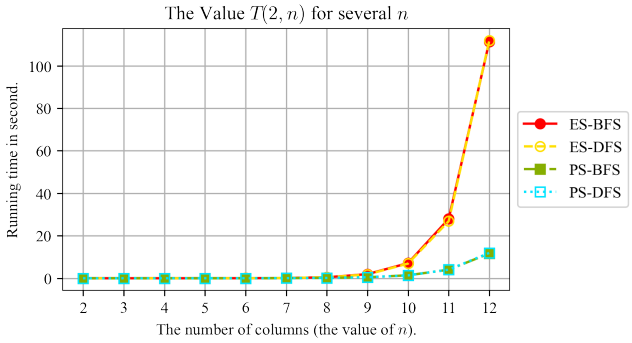


Fig. 18: The average running time for finding the number of solutions to empty instances of $2 \times n$ Yin-Yang puzzles where $n \in [2, 12]$.

Fig. 19 shows the average running time for finding the number of solutions to empty instances of $3 \times n$ Yin-Yang puzzle, where $3 \leq n \leq 8$. Here, we notice some differences between the average running time of the regular exhaustive search algorithm and its prune-and-search alternative for $n = 7$ and $n = 8$. This condition is similar to the previous result explained in Fig. 18.

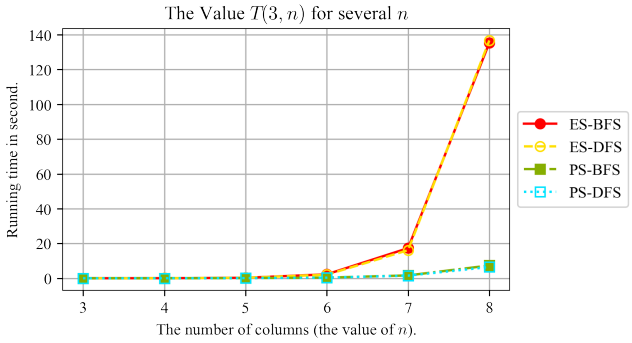


Fig. 19: The average running time for finding the number of solutions to empty instances of $3 \times n$ Yin-Yang puzzles where $n \in [3, 8]$.

We also conduct experiments to determine the average running time for finding the number of solutions to empty instances of $4 \times n$ and $5 \times n$ Ying-Yang puzzles. However, due to technical limitations of our computational environment, we only investigate the average running time for finding the number of solutions of 4×4, 4×5, 4×6, 5×5, and 5×6 puzzles. The average running times for finding the number of solutions to empty instances of 4×4, 4×5, and 4×6 puzzles are summarized in Table II, while the same measurement for the empty instances of 5×5 and 5×6 puzzles are summarized in Table III. In addition, due to the technical limitation of our computational device and C++ specification, we omit the experiment to determine the running time of the regular exhaustive search algorithm for finding the number of solutions to the 5×6 puzzle. The result of our experiment shows that—in terms of average running time—the prune-and-search technique significantly outperforms the exhaustive search approach for finding the solutions to the Yin-Yang puzzle. Here, the speed-up factor is non-linear, and the prune-and-search algorithm can find the solutions up to almost 50 times faster than its regular exhaustive search counterpart. Moreover, we see that there is no significant difference between the utilization of BFS-based and DFS-based verifications for each of the approaches.

TABLE II

The average running time (in seconds) for finding the number of solutions to empty instances of $4 \times n$ Yin-Yang puzzles where $n \in [4, 6]$.

Algorithm \ n	4	5	6
ES-BFS	0.6244	9.9965	168.3790
ES-DFS	0.6306	9.3669	167.6080
PS-BFS	0.1401	0.9648	6.7656
PS-DFS	0.1235	0.8883	6.0231

TABLE III

The average running time (in seconds) for finding the number of solutions to empty instances of $5 \times n$ Yin-Yang puzzle where $n \in [5, 6]$.

Algorithm \ n	5	6
ES-BFS	499.2820	–
ES-DFS	498.3727	–
PS-BFS	10.9921	116.3487
PS-DFS	11.0697	114.2330

VI. Concluding Remarks

We have discussed two algorithms for solving arbitrary Yin-Yang puzzles, namely the exhaustive search approach and the prune-and-search technique. In Section IV, we show that both algorithms use an $O(mn)$ time verification procedure for checking whether an $m \times n$ Yin-Yang configuration is also a solution. Moreover, both algorithms have an identical asymptotic running time of $O(\max\{mn, 2^{mn-h}\})$ for finding all solutions of a Yin-Yang instance with h hints of size $m \times n$. Nevertheless, our experiments show that, even though the asymptotic time complexity of both algorithms is identical, the prune-and-search technique practically outperforms the conventional exhaustive search approach for solving an $m \times n$ Yin-Yang puzzle.

The investigation regarding the mathematical and computational aspects of the Yin-Yang puzzle is still new and limited. We provide a function $S(m, n)$ denoting the number of solutions to an empty Yin-Yang instance of size $m \times n$ for $1 \leq m \leq n \leq 6$. However, this quantity does not consider the cases in which we may have an identical solution after we apply reflection or rotation. A more thorough investigation regarding the number of different solutions—up to reflection,

rotation, or a combination of both operations—is theoretically interesting. We suggest the application of Burnside's Lemma or Polya Enumeration Theorem to solve such a problem [36].

Finally, we suggest another exploration for solving Yin-Yang puzzles. Since Yin-Yang puzzles are NP-complete, then it is inherently interesting to construct a SAT-solver-based algorithm for solving such puzzles. This solver can be used to solve another open problem, such as determining the minimum number of hints required to ensure that an $m \times n$ Yin-Yang puzzle has a unique solution.

REFERENCES

[1] 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.

[2] G. Moore, *Paper, Pencil & You: Calm: Relaxing Brain-Training Puzzles for Stressed-Out People*. Greenfinch, 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] 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.

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

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

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

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

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

[12] 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.

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

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

[15] 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.

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

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

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

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

[20] 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

[21] 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.

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

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

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

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

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

[27] E. D. Demaine and M. Rudoy, "Tree-residue vertex-breaking: a new tool for proving hardness," in *Proceedings of the 16th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT2018)*, 2018.

[28] B. Bollobás, *Modern graph theory*. Springer Science & Business Media, 1998, vol. 184.

[29] R. Diestel, "Graph theory 3rd ed," *Graduate texts in mathematics*, vol. 173, p. 33, 2005.

[30] K. H. Rosen, *Discrete Mathematics and Its Applications, 8th Edtion*, 5th ed. McGraw-Hill Higher Education, 2019.

[31] Puzzling Stack Exchange, "How to prove yin-yang alternating 2 by 2 is not allowed," https://puzzling.stackexchange.com/questions/115722/how-to-prove-yin-yang-alternating-2-by-2-is-not-allowed, Apr. 2022, accessed: 28-4-2022.

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

[33] N. Megiddo, "Linear-Time Algorithms for Linear Programming in $\mathbb{R}^3$ and Related Problems," *SIAM journal on computing*, vol. 12, no. 4, pp. 759–776, 1983.

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

[35] E. D. Demaine, "Yin-Yang Font," https://github.com/edemaine/font-yinyang, Apr. 2022, accessed: 2022-05-11.

[36] M. Baxter, "The burnside di-lemma: combinatorics and puzzle symmetry," *Tribute to a Mathemagician*, pp. 199–210, 2005.