

RESEARCH ARTICLE

## List coloring based algorithm for the Futoshiki puzzle

Banu Baklan Şen <sup>\*</sup>, Öznur Yaşar Diner

Computer Engineering Department, Kadir Has University, Turkey  
banu.baklan@stu.khas.edu.tr, oznur.yasar@khas.edu.tr

### ARTICLE INFO

#### Article History:

Received 18 July 2023

Accepted 11 September 2024

Available Online 9 October 2024

#### Keywords:

List coloring

Precoloring extension

Latin square completion puzzle

Futoshiki puzzle

Personnel scheduling

AMS Classification 2010:

90C27; 05C85; 68Q25

### ABSTRACT

Given a graph  $G = (V, E)$  and a list of available colors  $L(v)$  for each vertex  $v \in V$ , where  $L(v) \subseteq \{1, 2, \dots, k\}$ , LIST  $k$ -COLORING refers to the problem of assigning colors to the vertices of  $G$  so that each vertex receives a color from its own list and no two neighboring vertices receive the same color. The decision version of the problem, LIST  $k$ -COLORING, is NP-complete even for bipartite graphs. As an application of list coloring problem we are interested in the Futoshiki Problem. Futoshiki is an NP-complete Latin Square Completion Type Puzzle. Considering Futoshiki puzzle as a constraint satisfaction problem, we first give a list coloring based algorithm for it which is efficient for small boards of fixed size. To thoroughly investigate the efficiency of our algorithm in comparison with a proposed backtracking-based algorithm, we conducted a substantial number of computational experiments at different difficulty levels, considering varying numbers of inequality constraints and given values. Our results from the extensive range of experiments indicate that the list coloring-based algorithm is much more efficient.



## 1. Introduction

Since the 1980s, there has been significant theoretical analysis and exploration of applications for pencil puzzle games. In recent decades, research has focused on algorithmic solutions and the computational complexity of pencil puzzle games, including optimization versions of various puzzle types. Latin Square Completion-Type Puzzles (LSCP) are among the most common types of these games.

A Latin Square Completion Puzzle (LSCP) is a partial Latin square with empty cells. A Partial Latin Square (PLS) is an  $n \times n$  grid that is partially filled with some numbers from  $[n] = \{1, \dots, n\}$ . The goal is to fill in all the blank cells with numbers in such a way that the numbers are distinct in each row and each column. The objective of LSCP is to complete the grid by filling the remaining cells with numbers such that each number appears exactly once in

each row and each column. Two notable puzzles in this category are Sudoku and Futoshiki.

The Futoshiki puzzle, also known as Unequal, is a popular Japanese board-based puzzle played on an  $n \times n$  square board with additional inequality constraints between certain cells. The objective is to fill the cells with numbers, satisfying the Latin square property while respecting the specified inequalities. Inequalities can occur between horizontally or vertically neighboring cells, indicating that a number in a particular cell must be greater or smaller than the number in the adjacent cell. Let  $S$  denote the set of inequality constraints and  $T$  the set of pre-assigned cells.

The decision version of the Futoshiki game, known as the FUTOSHIKI PROBLEM, is defined as follows:

#### FUTOSHIKI PROBLEM (FUTOSHIKI)

*Instance:*  $\mathcal{F}_n(T, S)$ , an  $n \times n$  board, a set  $T$  of

<sup>\*</sup>Corresponding Author

pre-assigned cells, and a set  $S$  of inequality constraints.

*Question:* Is the Futoshiki puzzle solvable on  $\mathcal{F}_n(T, S)$ ?

The solvability of a partial Latin square is closely related to Hall's condition. However, Bobga et al. [1] demonstrated that satisfying Hall's condition is insufficient. They provided necessary and sufficient conditions on the configuration of the prescribed cells to ensure the solvability of LSCP. Further results on related topics, such as partial Latinized rectangles, can be found in [2] and [3].

Both the decision version of the Latin Square completion problem and the FUTOSHIKI PROBLEM have been proven to be NP-Complete [4, 5].

Let us define the optimization version of FUTOSHIKI PROBLEM.

#### MAXIMUM FUTOSHIKI (MAXFUTOSHIKI)

*Input:*  $\mathcal{F}_n(T, S)$  and sign set  $S \subseteq S_L$ .

*Output:* A Futoshiki board  $\mathcal{F}_n(T, S)$  filled with maximum number of valid entries.

Various studies examine the computational complexity of problems defined on partial latin squares. The Latin Square completion problem is NP-Complete by reduction from 3-SAT [4]. In particular the Futoshiki problem is also known to be NP-Complete, as proved by Haraguchi et al. [5]. As for the optimization version of the Futoshiki, Haraguchi and Ono [5] examined the approximability of LSCPs and formulated three LSCP puzzles as maximization problems, presenting polynomial-time approximation algorithms. These maximization problems aim to fill as many cells as possible, instead of determining whether it is possible to complete the entire board. MAXFUTOSHIKI was shown to be NP-Hard by Haraguchi [5], and related work on optimization versions of LSCP problems is reviewed by Donovan [6].

Properties of Latin squares and improvements to Galvin's solution [7] have been explored by Ivanyi and Nemeth [8]. Yato and Seta [9] investigated the computational complexity and completeness of finding alternative solutions to LSCP problems and proposed two algorithms.

The solvability of LSCP puzzles has been extensively studied in terms of time complexity, and numerous algorithmic solutions have been proposed. Sudoku, a well-known puzzle, has been approached using various algorithmic techniques for both deterministic and metaheuristic approaches.

A deterministic algorithm does not contain any randomness or probabilistic elements. It always produces the same output and follows a fixed sequence of steps. Some major deterministic approaches to solve LSCP type problems are the exact cover problem with, Norvig's work with constraint propagation [10] and constraint programming that Crawford gave [11]. As for nondeterministic approaches, one can refer to the "Dancing Links" algorithm that Knuth presented [12].

The term "Metaheuristic" was first used in the study of Glover [13]. Metaheuristics are known as one of the best methods for finding sufficiently good solutions to NP-Hard problems. Traveling salesman problems, scheduling problems, and assignment problems are some of the examples that metaheuristics are used. Sudoku has been solved with one of the metaheuristic methods that are artificial bee colony algorithm [14], particle swarm optimisation [15], and ant colony optimisation algorithm [16]. Moreover, as a heuristics, we can show the study of Musliu [17] that proposes a hybrid method for solving Sudoku.

In this work, we concentrate on deterministic approaches rather than solving the puzzle with metaheuristic methods. One of the most common techniques to solve PLSs as a combinatorial optimization problem is coloring [4]. Furthermore, many graph coloring variants have been utilized to solve LSCP puzzles. For instance, in [18], the Sudoku puzzle is shown to illustrate the precoloring extension problem [19]. Precoloring extension is a variant of the precoloring problem in which some vertices are precolored and others are assigned lists of allowed colors. Notice that the NP-Completeness of the list coloring problem for general graphs [20] and bipartite graphs [21] has been proven.

Our motivation for studying the list coloring approach for the Futoshiki puzzle game stems from the fact that while Sudoku has been extensively studied as a graph coloring problem, Futoshiki has not been analyzed in the same context. In this paper, we adapt the Futoshiki puzzle game to a new variant of the list coloring problem, which we refer to as the *list precoloring extension problem* (formally defined in Section 2). We propose a list precoloring extension algorithm and discuss its complexity.

The rest of the paper is organized as follows. Section 2 provides problem definitions. Section 3 explores applications related to the Futoshiki game. Section 4 establishes the equivalence between the

list precoloring extension instance and the Futoshiki problem instance. Section 5 presents an algorithm to solve the Futoshiki problem when the board size is fixed. Section 6 analyzes the experimental results. Finally, Section 7 concludes the paper.

## 2. Problem definitions

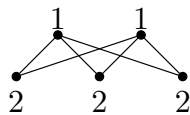
In this section, we provide formal notation and terminology related to graph coloring and introduce a new graph coloring problem that models the Futoshiki problem. The notation and basic terminology used in this section follow Dietel [22].

We consider simple, finite, undirected graphs  $G = (V, E)$  with a vertex set  $V$  and an edge set  $E$ . A *coloring* of a graph  $G$  is a labeling of its vertices. A *k-coloring* is a coloring that uses at most  $k$  colors from the set  $[k] = 1, 2, \dots, k$ . A coloring is *proper* if no two adjacent vertices have the same color. The decision version of the graph coloring problem is defined as follows:

### COLORING (COL)

*Instance:* A graph  $G = (V, E)$  and an integer  $k \geq 1$ .

*Question:* Does  $G$  have a  $k$ -coloring?



**Figure 1.** A graph  $G$  and a valid coloring for it.

In coloring,  $k$  is a part of the input. On the other hand, when  $k$  is fixed, i.e., when  $k$  is not a part of the input, we have the  $k$ -coloring problem. As an example, a 2-coloring is given in Figure 1: vertices in one part receive one color, while vertices in the other part receive a different color. It is worth noting that every bipartite graph can be colored using only two colors.

### k-COLORING (k-COL)

*Instance:* A graph  $G = (V, E)$ .

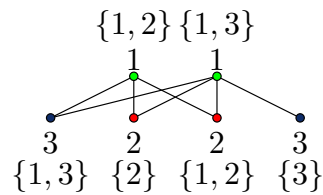
*Question:* Does  $G$  have a  $k$ -coloring?  $k$ -COL is NP-Complete for  $k \geq 3$  [23] and polynomial time solvable when  $k = 1$  or  $2$  [24]. *List coloring* is a generalization of graph coloring. It is a proper coloring in which each vertex  $v$  receives a color from its own list of allowed colors. The list coloring problem is defined by Vizing [23] and Erdős, Rubin and Taylor [25] independently.

### LIST-COLORING (LiCOL)

*Instance:* A graph  $G = (V, E)$  and a list assignment  $L$  for  $G$ .

*Question:* Does  $G$  have a coloring where each vertex  $v$  receives a color from its list  $L(v)$ ?

A *list assignment* of a graph  $G = (V, E)$  is a mapping  $L$  that assigns each vertex  $v \in V$  a List  $L(v) \subseteq \{1, 2, \dots\}$  of admissible colors for  $v$ . When  $L(v) \subseteq [k] = \{1, 2, \dots, k\}$  for every  $v \in V$  we say that  $L$  is a  $k$ -list assignment of  $G$ . Thus, the total number of available colors is bounded by  $k$  in a  $k$ -list assignment. On the other hand, when  $|L(v)| \leq k$  for every  $v \in V$ , then we say that  $L$  is a list  $k$ -assignment of  $G$ . Thus, the size of each list is bounded by  $k$  in a list  $k$ -assignment.



**Figure 2.** A list assignment  $L$  for the vertices of  $G$ , and a coloring that respects  $L$ .

The *List k-Coloring* problem is to decide whether a graph  $G = (V, E)$  with a list  $L(u) \subseteq \{1, \dots, k\}$  for each  $u \in V$  has a coloring  $c$  such that  $c(u) \in L(u)$  for every  $u \in V$ . It is clearly a generalization of  $k$ -coloring, and hence it is NP-Complete for  $k \geq 3$ . Refer to Figure 2 for an example. It is important to note that, despite the graph being bipartite in Figure 2, two colors were not enough to color it while satisfying the constraints imposed by the assigned lists  $L$ .

### LIST k-COLORING (Li k-COL)

*Instance:* A graph  $G = (V, E)$  and a  $k$ -list assignment  $L$ .

*Question:* Does  $G$  have a coloring where each vertex  $v$  receives a color from its list  $L(v)$ ?

A  $k$ -precoloring of a graph  $G = (V, E)$  is a mapping  $c_W : W \rightarrow \{1, 2, \dots, k\}$  for some subset  $W \subseteq V$ . We say that a  $k$ -coloring  $c$  of  $G$  is an *extension* or a *k-extension* of a  $k$ -precoloring  $c_W$  of  $G$  if  $c(v) = c_W(v)$  for each  $v \in W$ . For a given graph  $G$ , a positive integer  $k$  and a  $k$ -precoloring  $c_W$  of  $G$ , the Precoloring Extension problem (PREXT) asks whether  $c_W$  can be extended to a  $k$ -coloring of  $G$ . If  $k$  is fixed we denote this problem as the  $k$ -Precoloring Extension problem ( $k$ -PREXT). Let us define the latter problem formally.

**$k$ -PRECOLORING EXTENSION ( $k$ -PREXT)**

*Instance:* A graph  $G = (V, E)$  and a  $k$ -precoloring  $c_W$ .

*Question:* Is there a  $k$ -extension for  $c_W$ ?

For general graphs PREXT is NP-Complete [26]. In fact, the NP-Completeness of the LSCP problem is shown via its equivalence to the  $k$ -PREXT when it is restricted to the cartesian product of  $K_n$  with itself [4].

We define a new coloring problem called the List  $k$ -Precoloring Extension problem (LI  $k$ -PREXT).

**LIST  $k$ -PRECOLORING EXTENSION (LI  $k$ -PREXT)**

*Instance:* A graph  $G = (V, E)$ ,  $W \subseteq V$ , a  $k$ -precoloring  $c_W$ , and a list  $k$ -assignment  $L$  for each  $v \in V/W$

*Question:* Is there a  $k$ -extension for  $c_W$  that obeys the list  $L$ ?

Notice that when the list  $L$  is not assigned to the vertices in  $V/W$ , then LI  $k$ -PREXT reduces to  $k$ -PREXT. Let us denote an instance of LI  $k$ -PREXT with  $\mathcal{L}_G(c_W, L)$ .

### 3. Applications

In this section, we will first provide a brief overview of some notable applications related to the problems under consideration, namely the Futoshiki problem, list coloring, and its variants. Subsequently, we will introduce a novel application of the Futoshiki problem in the field of scheduling, specifically to optimize the efficiency of the job assignment problem.

**Applications of the Futoshiki problem:** The Futoshiki problem has found applications in various domains. Mahmood [27] proposed a random number generator that utilizes the Futoshiki problem to generate numbers satisfying given conditions. This generator, with good linear complexity, has potential application as an encryption key in mathematical analysis, security systems, and simulations. Additionally, Haraguchi [28] explored the evaluation values achievable in a Futoshiki puzzle with a high number of inequality signs.

The Futoshiki configuration technique, considering partial shading conditions in photovoltaic (PV) systems, has been proposed by Sahu et al. [29]. They observed that incorporating the Futoshiki structure increases the power generation of PV arrays, leading to improved energy efficiency. The technique avoids the need for changing the electrical connection of modules by rearranging them, and it effectively reduces mismatch loss under different shading models.

**Applications of List Coloring and its Variants:**

The list coloring problem has been widely applied to solve optimization and scheduling problems [30]. In Orden and Moreira's work [31], the problem of minimizing interference threshold and the number of colors respecting that threshold was modeled as list coloring. They demonstrated that the problems are NP-Hard and proposed DSATUR, a graph coloring algorithm, to tackle them.

Garg et al. [32] tackled the channel frequency allocation problem in mobile communication networks by modeling it as a generalized list coloring problem. Their solutions prevented signal interference by selecting channels for neighboring base stations in a way that they did not overlap. This approach effectively addressed the crash failures caused by distance limitations.

In the domain of register assignment, Zeitlhofer et al. [33] presented a list-coloring algorithm that optimally assigns a large number of target variables to a small number of CPU registers. This algorithm preserves the structure of the interference graph, ensuring the retention of interval graph properties.

Sudoku puzzles can also be formulated as list coloring problems. Each cell corresponds to a vertex, and the relationships between cells are represented as edges in rows and columns. The numbers used in Sudoku can only appear once in each row and column, making it an instance of the list coloring problem. Additionally, Lastrina et al. [18] demonstrated how the precoloring extension problem can be used to illustrate the Sudoku puzzle.

### 4. LI $k$ -PREXT and FUTOSHIKI

In this section, we show that the Futoshiki problem can be reduced to the list precoloring extension problem for the Futoshiki graph  $G$ . In the reduction the revealed cells given in the Futoshiki problem are used to construct the pre-coloring for  $G$ . In addition, the list assignment is obtained using the inequality constraints.

Let  $n \geq 2$  be a positive integer. A Partial Latin Square is an  $n \times n$  grid that is partially filled with some numbers from  $[k] = \{1, \dots, n\}$ . Let us denote a cell that is in the  $i$ 'th row and the  $j$ 'th column of a grid as  $(i, j)$ . Each cell  $(i, j)$  is represented in the graph with a vertex  $v_{ij}$ . Two cells  $(i, j)$  and  $(i', j')$  are adjacent whenever they are in the same column or in the same row.

Thus, a Latin square is represented as a graph  $G = (V, E)$  such that  $V = \{v_{ij} : 1 \leq i, j \leq n\}$ , and  $E = \{(v_{ij}, v_{i'j'}) : ((i = i') \wedge (j \neq j')) \vee ((j = j') \wedge (i \neq i'))\}$  [34]. This graph is called the *Futoshiki graph of size  $n$* . Notice that the Futoshiki graph  $G$  has  $n^2(n - 1)$  edges. The graph  $G$  is isomorphic to the graph  $K_n \boxtimes K_n$ , which is the strong product of  $K_n$  with itself [18] and it is  $(2n - 2)$ -regular.

Recall that the Futoshiki problem of size  $n$ ,  $\mathcal{F}_n(T, S)$ , is defined on the Futoshiki graph of size  $n$  where  $S$  is the set of inequality constraints. Thus there are at most  $2n(n - 1)$  inequality signs. In Figure 3, for  $n = 4$ , there are  $n^2 = 16$  vertices and  $n^2(n - 1) = 48$  edges. An instance of the problem on this graph can take up to 24 inequality constraints in total, yet in this instance, there are only 5 inequality constraints.

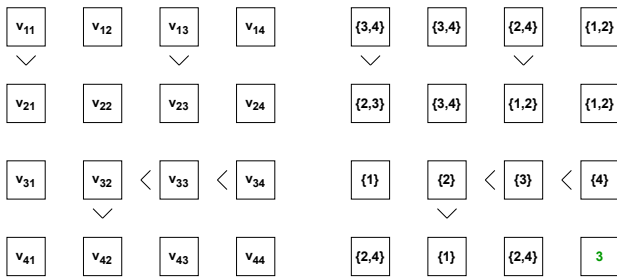


Figure 3. Vertices of the Futoshiki Puzzle.

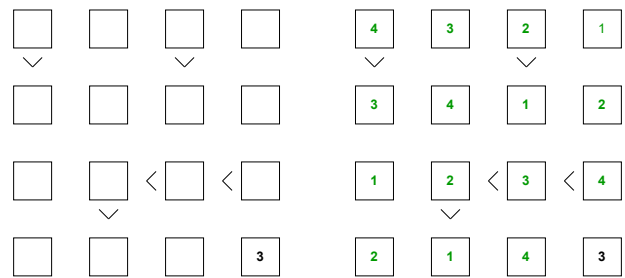
**Theorem 1.** For every Futoshiki problem  $\mathcal{F}_n(T, S)$  there exists an equivalent instance of the LIST  $k$ -PRECOLORING EXTENSION problem  $\mathcal{L}_G(c_W, L)$  on the Futoshiki graph where  $k = n$ .

**Proof.** We give a polynomial time reduction that converts a Futoshiki problem  $\mathcal{F}_n(T, S)$  of size  $n \times n$  to a Futoshiki graph  $G$  and a list-assignment  $L$  that corresponds to the list  $k$ -precoloring extension problem  $\mathcal{L}_G(c_W, L)$  for some precoloring  $c_W$ . We also show that  $\mathcal{F}_n(T, S)$  is solvable whenever  $\mathcal{L}_G(c_W, L)$  is solvable on  $G$ . We do the latter by converting each solution of  $\mathcal{F}_n(T, S)$  to a solution of  $\mathcal{L}_G(c_W, L)$  and vice versa.

Given an instance  $\mathcal{F}_n(T, S)$  of the Futoshiki problem the entries of the  $n \times n$  board cells correspond to vertices of  $G$ , occurring exactly once in each row and column. Thus, the graph  $G$  will have  $n^2$  vertices. The cells represent vertices and adjacent cells in each row and column represent the edges. At the beginning of the problem, if a number  $l$  is revealed in a cell that is represented with a vertex  $v$ , then we say  $v$  is precolored with color  $l$ . This gives a one-to-one correspondence between the revealed cells  $T$  and the precoloring

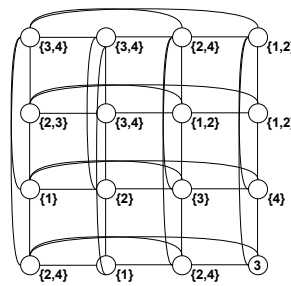
$c_W$ . If no number is revealed in the cell, then the corresponding vertex will be assigned the list  $\{1, 2, \dots, k\}$ . If, in addition, there is an inequality sign  $>$  located between some adjacent cells that are represented with vertices  $u$  and  $v$  in  $G$ , then for every color  $i \in L(u)$ , there must be at least one color  $j \in L(v)$  such that  $i > j$ . For the inequality sign  $<$ , the construction of the list assignments of the related vertices are done similarly. This gives the construction of the list assignment  $L$ , thereby completing the reduction of  $\mathcal{F}_n(T, S)$  to  $\mathcal{L}_G(c_W, L)$ .

Each solution of  $\mathcal{F}_n(T, S)$  will naturally give a proper coloring for  $G$  which is an extension of the precoloring  $c_W$  and it will obey the list  $L$ . On the other hand, a list coloring solution of  $\mathcal{L}_G(c_W, L)$  yields a solution to the given Futoshiki problem  $\mathcal{F}_n(T, S)$ . Notice that the precoloring and the list assignment  $L$  are constructed so that the solution to the list precoloring extension problem gives a number assignment that satisfies the inequalities located between adjacent cells. □

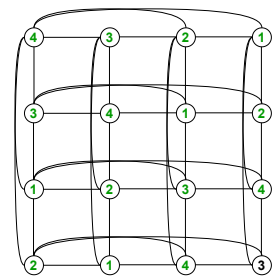


a) Futoshiki puzzle game

b) Solution to Futoshiki puzzle game



c) List precoloring extension



d) Coloring

Figure 4. A Futoshiki puzzle instance and the cells that correspond to the vertices in the related Futoshiki Graph.

Figure 4 illustrates an instance of a  $4 \times 4$  Futoshiki problem and the corresponding Futoshiki graph construction along with the list assignment: Figure 4.a shows the initial board of the game, Figure 4.b shows the solution of the game, Figure 4.c

shows the list precoloring extension instance and, finally, Figure 4.d shows the corresponding coloring as a solution to the list precoloring extension problem.

## 5. List coloring-based algorithm for the FUTOSHIKI PROBLEM

There are various algorithmic approaches to solve pencil puzzles. The backtracking algorithm is used for the class of Constraint Satisfaction Problems (CSPs). These problems are defined as a set of variables, a set of their respective domains of values, and a set of constraints [10]. The goal of a CSP is to find a consistent assignment of values to variables that satisfies all constraints, subject to certain conditions. CSPs have applications in various domains, including scheduling, planning, configuration, and puzzle-solving. LSCP-type puzzles are seen as constraint satisfaction problem CSPs that find a solution that satisfies all the constraints considering assignment of variables. These problems include Sudoku, Futoshiki, and Kakuro. Sudoku is the most popular one that is solved as CSP by the researchers. Norvig [10] describes two methods to solve Sudoku, namely, constraint propagation (CP) and Local Search. Constraint propagation is a technique that is commonly used in CSPs to efficiently update and reduce the domain of variables based on the constraints imposed by the problem.

In this section, we present two different deterministic algorithms for solving the Futoshiki Puzzle, called FutoshikiBT and ColorFutoshik, each incorporating backtracking, and filtering methods respectively.

### 5.1. BackTracking algorithm

The backtracking algorithm can be seen as the simplest solution for Sudoku puzzles which are the most commonly studied problem. Backtracking uses a recursive approach in which each cell is assigned a number from  $1 \dots n$  when the board size is  $n \times n$ . The backtracking algorithm systematically explores the solution space by iteratively assigning values to empty cells in the puzzle and backtracking when a contradiction or violation of constraints is encountered.

Here, we give a backtracking algorithm that solves the Futoshiki Puzzle to compare its efficiency with the proposed method called ColorFutoshiki which we give in Section 5.2. The backtracking algorithm starts by selecting an empty cell in the puzzle and attempts to assign a value that satisfies

the row and column constraints, as well as the inequality constraints associated with neighboring cells. It then moves on to the next empty cell and repeats the process. If a contradiction arises, such as a repeated number in a row or column, or a violation of an inequality constraint, the algorithm backtracks to the previous cell and explores alternative value assignments. Although this method guarantees a solution, it is not efficient in terms of time complexity. Let us present our FutoshikiBT Algorithm.

---

### Algorithm 1 FutoshikiBT Algorithm

---

```

1: Input: Futoshiki board with constraints.
2: Output: Solution of the puzzle.

3: if  $colorG(n, list, v = 1, given) == False$ 
4:   print ("No solution")
5: else
6:   print (list)
7:  $colorG(n, list, v, given)$ :
8:   if  $(v == V + 1)$ 
9:     return True
10:  for  $c$  in range(1,  $n$ )
11:    if  $safe(v, list, c, given) == True$ 
12:      list[ $v$ ] =  $c$ 
13:      if  $colorG(n, list, v + 1, given)$ 
14:        return True
15:      end if
16:      if  $v$  not in  $given$ 
17:        list[ $v$ ] = 0
18:      end if
19:      return False
20:    end if
21:  end for
22:  return False
23:  $safe(v, list, c, given)$ :
24:  if  $v$  in  $given$  and  $list[v] == c$ 
25:    return True
26:  else if  $v$  in  $given$ 
27:    return False
28:  end if
29:  for  $(i$  in range(1,  $V$ ))
30:    if  $list[i] == c$  and  $neighbour(v, i)$ 
31:      return False
32:    end if
33:    if constraints are not satisfied
34:      return False
35:    end if
36:  end for
37:  return True

```

---

## 5.2. ColorFutoshiki algorithm

In this section, using the equivalence between the FUTOSHIKI PROBLEM and the LI  $k$ -PREXT problem for the Futoshiki graph, as assured by Theorem 1, we construct the ColorFutoshiki algorithm to solve the FUTOSHIKI PROBLEM. We will observe that, this approach is equivalent to the backtracking algorithm with forward checking for the FUTOSHIKI PROBLEM.

The backtracking algorithm considers every solution by iterating every possible number in each cell under all satisfying conditions, assigns the first available option, backtracks when a solution is not possible for the next cell under consideration, and tries the next possible option for the previous cell. These methods guarantee the solution, but they do not give the solution in optimal time. Here we aim to improve the backtracking algorithm. This is why we need a problem space that helps our solver save us more time. The backtracking method of solving the Futoshiki problem fills each cells from left to right and top to bottom with considering inequality constraints.

The ColorFutoshiki algorithm is an improved version of the FutoshikiBT algorithm. Our aim is to reduce the number of colors in each list by eliminating inconsistent ones. This reduces the search space to be explored. At the beginning of the ColorFutoshiki algorithm, we use a filtering technique. In this filtering step we do forward checking in order to create color lists. Forward checking keeps track of the remaining possible values for unassigned variables after a variable is assigned a value. It propagates constraints by eliminating values from the domains of other variables that conflict with the newly assigned value. This technique is applied to create lists for each cell that they can use. Thus, it will begin coloring the puzzle with the minimum number of colors in the list for each cell.

## 5.3. Analysis of the algorithms

The input parameters for the algorithm are the Futoshiki graph of size  $n$ , the inequality constraints, and the pre-assigned entries. The algorithm begins by examining the constraints and pre-assigned numbers of the Futoshiki instance  $\mathcal{F}_n(T, S)$ , which then produces a  $k$ -precoloring instance  $\mathcal{LG}(c_W, L)$ . Next, it determines how to color the given Futoshiki graph using the list of colors assigned to each vertex through the reduction process described above. If  $\mathcal{LG}(c_W, L)$  is a YES instance, the algorithm outputs a matrix

$M_G = [m_{ij}]n \times n$  indicating the colors of the vertices of the graph  $G$ . Otherwise, it concludes that no solution exists.

First, let us analyze the FutoshikiBT algorithm, which builds candidates for the solutions incrementally and abandons candidates when it determines that they cannot possibly be solved with a valid solution.

The function  $\text{colorG}(n, \text{list}, v, \text{given})$  is a recursive function that ensures the coloring process is completed by checking all vertices. It attempts to use the colors in the list for the corresponding vertex in order. Here,  $n$  represents the puzzle dimension, which is equal to the size of the color list.  $\text{list}$  is the list of colors assigned to vertices (solution).  $v$  is the vertex number.  $V$  represents the total number of vertices.  $\text{given}$  denotes the values given before the game starts.

The function  $\text{safe}(v, \text{list}, c, \text{given})$  checks whether the given vertex can be colored with the chosen color by verifying the constraints. In this process,  $\text{neighbour}(v, i)$  checks the adjacency of the two relevant vertices, ensuring that adjacent vertices are not colored with the same color.

It is worth noting that the FutoshikiBT algorithm does not include a process for creating preassigned color lists. On the other hand, ColorFutoshiki first traverses the graph and creates a list of colors that minimizes the number of candidate colors for each cell. It then attempts to color empty cells, starting from the first vertex  $v_{11}$ .

The ColorFutoshiki algorithm is an improved version of the FutoshikiBT algorithm. Its aim is to reduce the number of colors in each list by eliminating inconsistent ones. This reduction effectively reduces the search space that needs to be explored.

The pseudocode of the algorithm is provided below.

**Algorithm 2** ColorFutoshiki Algorithm

---

```

1: Input: Futoshiki board with constraints.
2: Output: Solution of the puzzle.

3: if  $colorG(pcList[1].length, list, v = 1, given)$ 
4:   print (list)
5: else
6:   print ("No solution")
7: end if

8:  $colorG(n, list, v, given)$ :
9:   if  $(v == V + 1)$ 
10:    return True
11:   end if
12:   for  $c$  in  $range(1, n)$ 
13:     if  $safe(v, list, pcList[v][c], given)$ 
14:        $list[i] = pcList[v][c]$ 
15:        $nextN = pcList[v + 1].length$ 
16:       if  $colorG(nextN, list, v + 1, given)$ 
17:         return True
18:       end if
19:       if  $v$  not in  $given$ 
20:          $list[v] = 0$ 
21:       end if
22:       return False
23:     end if
24:   end for
25:   return False

26:  $safe(v, list, c, given)$ :
27:   if  $v$  in  $given$  and  $list[v] == c$ 
28:     return True
29:   else if  $v$  in  $given$ 
30:     return False
31:   end if
32:   if constraints are not satisfied
33:     return False
34:   end if
35:   for  $i$  in  $range(1, n)$ 
36:     if  $(hNeighbor == c \ \& \ hNeighbor! = v)$ 
37:       return False
38:     end if
39:     if  $(vNeighbor == c \ \& \ vNeighbor! = v)$ 
40:       return False
41:     end if
42:   end for
43:   return True

```

---

In the ColorFutoshiki algorithm, first, we perform filtering and create color lists for each cell based on their admissible colors and constraints. Consequently, the algorithm begins coloring the puzzle using the minimum number of colors available in the lists.

Unlike the FutoshikiBT algorithm, ColorFutoshiki uses a recursive structure that does

not traverse the entire graph to color the related vertex. Instead, it only checks the horizontal and vertical neighbors of the vertex being colored. It is observed that ColorFutoshiki outperforms the FutoshikiBT algorithm in all instances of varying difficulty levels.

Now, let us explain the ColorFutoshiki algorithm.

In lines 3-7, we call the colorG function, which displays the solution if found. Here pcList is the list that each cell takes.

In lines 8-25, the recursive colorG function checks the termination condition. If this condition is not met, it checks whether the related vertex  $v$  can be colored based on the possible color list of  $v$ . If  $v$  cannot be colored, the algorithm moves on to the next color in its list. If it cannot be colored with any color in the list, the recursive function returns to previous vertex and the color of the previous vertex is updated. If it can be colored, the algorithm moves on to the next vertex, and the admissible color is added to the color list.

In lines 26-43, if the vertex to be colored has a preassigned (given) value, the algorithm proceeds to the next vertex. Then, it checks whether there is a constraint in front of or above the vertex to be colored or not. If there is, it verifies whether the constraint conditions are satisfied. After this step, we check the colors of the adjacent vertices if they have the same color. Instead of traversing all nodes, it only checks the horizontal and vertical neighbors of the relevant vertex.

Now let us analyze the time complexities of the algorithms. The FutoshikiBT algorithm traverses all vertices to check the neighborhood of the vertex being colored and to decide whether the colors are the same or not. For each empty cell, there are  $n$  possible options, where  $n$  is the total number of colors. As a result, the time complexity becomes  $O(n^{n^2})$ . In the ColorFutoshiki algorithm some values are removed from some domains. Since there will be some early pruning the time taken will be much less than the backtracking algorithm. However, the upper bound time complexity remains the same. The reason is that we don't know how many values are removed. As a result, the time complexity becomes  $O(n^{n^2})$ .

Although the worst case time complexity of the ColorFutoshiki algorithm is only a slight improvement over the FutoshikiBT algorithm, as we will observe below, its speed is remarkably faster in all of the computational experiments that we have done.



## 6. Results and discussion

We aim to solve the FUTOSHIKI as an instance of the LICOL problem. For this reason we first improved the **ColorFutoshiki** algorithm. It is an enumeration algorithm that incorporates additional search space reductions and bounding elements, making it an enhanced version compared to FutoshikiBT. Due to the fact that FUTOSHIKI is classified as an NP-Complete problem, ColorFutoshiki proves to be highly effective and suitable for numerous applications and purposes. Its capabilities make it a valuable tool in addressing the complexity of FUTOSHIKI.

Here, we present the computational experiments conducted to assess the efficiency of the ColorFutoshiki algorithm in solving instances of the Futoshiki problem. All the codes were implemented in the Python programming language, and the experiments were executed on a system with an Intel Core i7-6700HQ CPU operating at 2.60 GHz, with 16GB RAM, running Windows 10 (64-bit). Different instances have been generated and tested on  $n \times n$  boards for  $n = 6, 7, 8, 9, 15, 20, 30, 40, 50$ . We ran the related code 50 times per instance and took their average. In total, 1800 Futoshiki puzzles have been used for each algorithms. Due to space limitations, we are unable to provide a table containing the running times for each individual test. This is why, we present the average experimental results for each algorithm in Table 1 and Table 2.

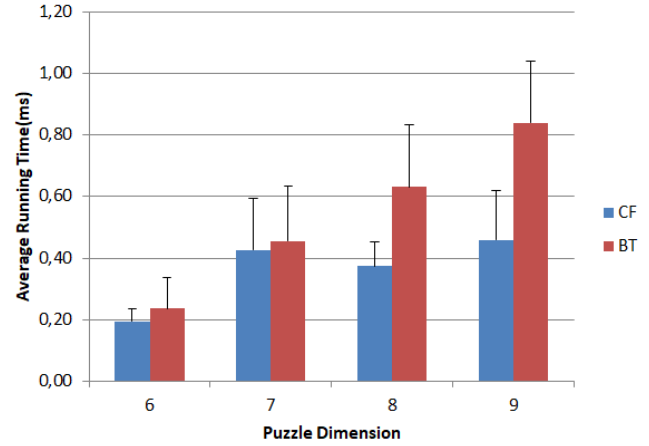
Solutions to the FUTOSHIKI Problem are of particular interest due to the given application in scheduling on  $n \times n$  boards when the size  $n$  is large. As our experimental results demonstrate, the ColorFutoshiki algorithm works much better even on boards of a larger size where the number of inequality constraints is not necessarily restricted to be less than  $n$  for an  $n \times n$  board.

A standard deviation denotes the spread of data concerning its mean. When the standard deviation is small, it signifies that the data is tightly clustered around the mean. Conversely, a high or large standard deviation suggests that the data is more widely spread.

All the results and standard deviations can be seen in Table 1. Different numbers of constraints and different numbers of givens are examined for each algorithm.

The performance of the proposed algorithms for standard search algorithms is illustrated in Figure 5 and Figure 6 with standard deviations. In this figures, we maintain a constant number of constraints while showcasing the increasing number

of given values. Notably, ColorFutoshiki consistently outperforms FutoshikiBT in solving puzzles, even when the number of constraints is held constant. These results underscore the efficiency of the ColorFutoshiki algorithm, particularly on larger-sized boards, providing a performance comparison with the FutoshikiBT algorithm.



**Figure 5.** Comparison of **ColorFutoshiki**(CF) and **FutoshikiBT**(BT) on smaller board sizes.

In addition to running times, we also measure the number of operations for both ColorFutoshiki and FutoshikiBT. Since the number of explored nodes provides insights into the efficiency of the algorithm, we show both the number of explored nodes and the number of removed values of each algorithm in Table 2. A lower number of explored nodes generally indicates a more efficient algorithm, as it suggests that the algorithm is able to reach a solution without exhaustively searching through a large portion of the puzzle's solution space. The number explored nodes allows for comparison with other algorithms or approaches for solving the same puzzle. For this reason, we use this parameter to compare the ColorFutoshiki algorithm that we present with FutoshikiBT.

We observe that the number of explored nodes varies significantly across different instances of the puzzle, it may indicate that the algorithm's performance is sensitive to certain characteristics of the puzzle. Similar to the number of explored nodes, the number of removed nodes provides insight into the efficiency of the algorithm. In certain search algorithms, such as backtracking or constraint satisfaction algorithms, removed nodes typically refer to nodes that are pruned from the search space because they are deemed unnecessary or invalid. A lower number of removed nodes indicates that the algorithm is effectively pruning the search space, which can lead to improved

**Table 1.** Run time for all algorithms reported in milliseconds.

size	inequalities	givens	CF	BT
6	6	1	1.54 ± 0.11	1.61 ± 0.44
6	6	20	0.28 ± 0.05	0.29 ± 0.1
6	6	30	0.19 ± 0.04	0.24 ± 0.1
6	1	6	0.59 ± 0.08	0.66 ± 0.22
6	20	6	0.5 ± 0.08	0.54 ± 0.27
6	30	6	0.53 ± 0.17	0.4323 ± 0.1073
7	7	1	1.40 ± 0.16	1.47 ± 0.52
7	7	30	0.43 ± 0.17	0.46 ± 0.18
7	7	40	0.29 ± 0.04	0.35 ± 0.08
7	1	7	1.37 ± 0.3	1.58 ± 0.5
7	30	7	2.07 ± 0.46	2.33 ± 0.58
7	40	7	1.19 ± 0.37	1.21 ± 0.52
8	8	1	3.53 ± 0.80	3.64 ± 1.33
8	8	40	0.5 ± 0.04	0.77 ± 0.29
8	8	50	0.37 ± 0.08	0.63 ± 0.2
8	1	8	2.01 ± 0.33	2.29 ± 0.73
8	40	8	6.72 ± 1.71	9.33 ± 2.32
8	50	8	5.34 ± 1.55	10.04 ± 2.3
9	9	1	8.08 ± 3.11	8.11 ± 3.06
9	9	70	0.46 ± 0.16	0.84 ± 0.2
9	18	35	1.28 ± 0.32	1.92 ± 0.63
9	1	9	3.78 ± 1.33	4.7 ± 1.69
9	70	9	8.86 ± 0.3	10.71 ± 0.92
9	35	18	2.27 ± 3.27	2.95 ± 3.22
15	15	150	2.77 ± 0.64	11.65 ± 2.54
15	15	170	2.25 ± 0.86	11.75 ± 2.86
15	15	200	1.44 ± 0.45	10.52 ± 3.94
15	1	150	11.82 ± 0.7	12.04 ± 3.26
15	90	130	3.47 ± 1.10	12.61 ± 3.33
15	150	15	2987.7 ± 336.8	3064.9 ± 103.2
20	20	300	4.98 ± 1.48	38.54 ± 8.12
20	30	320	11.58 ± 1.27	62.2 ± 3.43
20	20	370	1.575 ± 1.62	31.57 ± 6.07
20	30	300	5.21 ± 3.48	35.98 ± 8.52
20	80	250	349.97 ± 1.42	644.16 ± 7.26
20	50	350	3.55 ± 27.31	33.27 ± 29.87
30	30	500	11.000 ± 37.377	2585.5 ± 133.84
30	30	600	612.73 ± 34.274	1360.2 ± 441.69
30	30	750	41.852 ± 4.9579	1192.3 ± 78.302
30	50	500	1103.7 ± 135.63	2412.5 ± 102.47
30	100	600	608.17 ± 29.557	1274.6 ± 71.195
30	300	750	40.34 ± 5.8942	993.95 ± 34.361
40	40	750	2827.7 ± 97.233	12159 ± 402.91
40	40	950	702.37 ± 33.654	9869.8 ± 234.02
40	40	1300	91.642 ± 8.8446	1257.7 ± 51.235
40	100	750	2913.0 ± 599.59	9359.7 ± 255.39
40	300	950	437.39 ± 23.055	4845.2 ± 130.88
40	400	950	438.99 ± 27.05	4995.3 ± 142.66
50	50	1200	4443.1 ± 151.71	37727 ± 579.75
50	100	1500	2158.0 ± 93.02	21803.0 ± 358.64
50	250	2000	1817.7 ± 75.545	20849 ± 538.47
50	400	1600	1092.8 ± 54.712	13279 ± 162.22
50	400	2000	1067.5 ± 39.437	13145 ± 263.9
50	500	2000	835.95 ± 40.199	11559 ± 284.97

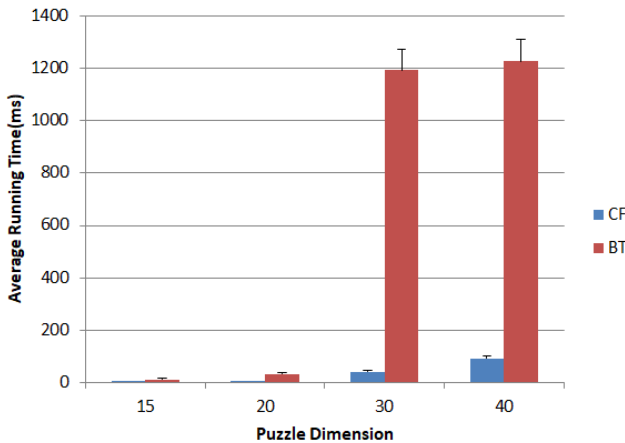
efficiency as it can be seen in Table 2. As the FutoshikiBT algorithm lacks a filtering step, no

values are removed from its domains, unlike the filtering steps in the ColorFutoshiki algorithm.

The running time of the algorithm varies depending on the puzzle's difficulty level. We observe that as the number of inequality signs approaches the maximum limit, the computation time significantly decreases. This behavior can be attributed to the utilization of the backtracking method in the ColorFutoshiki algorithm. Typically, Futoshiki puzzles are played on  $5 \times 5$  to  $9 \times 9$  boards (occasionally on  $15 \times 15$  boards), and existing algorithmic solutions are primarily tested on boards with dimensions up to  $n = 9$ .

In Figure 5, we compare the performance of our method with previous solutions employing the FutoshikiBT algorithm on smaller-sized boards.

Motivated by the lack of performance analysis for larger-sized boards, we conducted experiments using the ColorFutoshiki algorithm on larger board sizes. Additionally, we wanted to assess the algorithm's effectiveness on larger-sized boards due to the relationship between the Futoshiki puzzle game and larger scheduling problems, as discussed in Section 3. In Figure 6, we compare the performance of our method with previous solutions employing the FutoshikiBT algorithm on larger-sized boards. We even conducted experiments for  $50 \times 50$  boards. The results demonstrate that the ColorFutoshiki algorithm efficiently solves even  $30 \times 30$ ,  $40 \times 40$  and  $50 \times 50$  board games as shown in Figure 7.



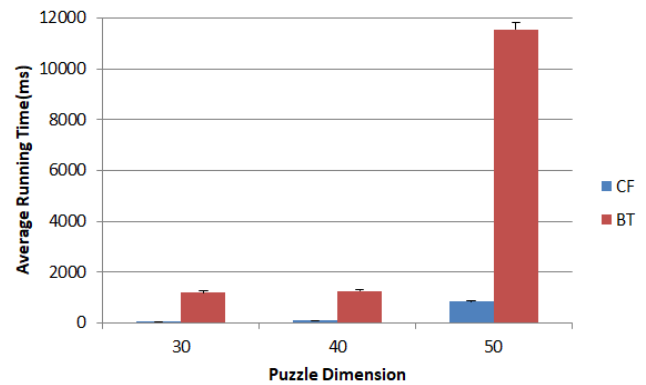
**Figure 6.** Comparison of ColorFutoshiki(CF) and FutoshikiBT(BT) on larger board sizes.

Overall, these findings highlight the efficiency of the ColorFutoshiki algorithm, especially on larger-sized boards, and provide a performance comparison with the FutoshikiBT algorithm.

## 7. Conclusion

The Futoshiki problem is aimed to be solved as a list coloring problem in ColorFutoshiki. It is an enumeration algorithm that incorporates additional search space reductions and bounding elements, making it an enhanced version compared to FutoshikiBT. Due to the fact that FUTOSHIKI is classified as an NP-Complete problem, ColorFutoshiki proves to be highly effective and suitable for numerous applications and purposes. Its capabilities make it a valuable tool in addressing the complexity of FUTOSHIKI.

A considerable number of experiments were conducted to test ColorFutoshiki and FutoshikiBT, providing a robust foundation for drawing meaningful conclusions. The extensive set of experiments carried out ensures that the findings are sufficiently supported and reliable. Observing Table 1 and Table 2, we see that ColorFutoshiki is much more efficient than FutoshikiBT.



**Figure 7.** Comparison of ColorFutoshiki(CF) and FutoshikiBT(BT) on larger board sizes.

We incorporate a short discussion on these metaheuristic methods and their applications to Sudoku. This will enhance the comprehensiveness of our paper and provide a broader perspective on the algorithmic techniques used to solve LSCP type puzzles.

We are also interested in studying the Futoshiki problem as an application of the LI  $k$ -COL problem. To find approximation algorithms for the Futoshiki puzzle, we would like to use metaheuristics.

As for future work related with nature based algorithms, one can see whether an Ant Colony optimization (ACO) algorithm gives a more efficient algorithm to solve the Futoshiki problem. ACO is a Swarm intelligence algorithm which is one of the artificial intelligence techniques. A solution

**Table 2.** The count of removed nodes and explored nodes. NRN stands for "No Removed Nodes".

size	inequalities	givens	FC-EN	FC-RN	BT-EN	BT-RN
6	6	1	528	15	540	NRN
6	6	20	40	173	126	NRN
6	6	30	36	180	126	NRN
6	1	6	151	78	198	NRN
6	30	6	111	81	156	NRN
6	20	6	100	81	144	NRN
7	7	1	456	18	469	NRN
7	7	30	59	276	196	NRN
7	7	40	50	291	196	NRN
7	1	7	298	113	406	NRN
7	40	7	268	125	392	NRN
7	30	7	474	124	392	NRN
8	8	1	1145	29	1160	NRN
8	8	50	65	446	288	NRN
8	8	40	78	423	288	NRN
8	1	8	580	156	736	NRN
8	40	8	1851	161	2744	NRN
8	50	8	1293	174	2920	NRN
9	9	1	2279	24	2313	NRN
9	9	70	81	647	405	NRN
9	18	35	187	546	603	NRN
9	1	9	988	194	1357	NRN
9	70	9	1629	213	2070	NRN
9	35	18	489	385	855	NRN
15	15	150	286	3040	2010	NRN
15	15	170	239	3102	1800	NRN
15	15	200	229	3139	1800	NRN
15	1	150	286	3028	2010	NRN
15	90	130	331	2975	2025	NRN
15	150	15	674261	7472	240	NRN
20	20	300	472	7589	4200	NRN
20	30	320	760	7484	5940	NRN
20	20	370	408	7588	4220	NRN
20	30	300	472	7484	4200	NRN
20	80	250	21057	6922	49580	NRN
20	50	350	417	7578	4220	NRN
30	30	500	44063	22402	116550	NRN
30	30	600	24682	23577	56400	NRN
30	30	750	1722	25450	48060	NRN
30	50	500	41511	22403	111840	NRN
30	100	600	22654	23580	52560	NRN
30	300	750	1684	25510	41220	NRN
40	40	750	69967	54498	229320	NRN
40	40	950	17714	56006	537760	NRN
40	40	1300	2542	61272	43160	NRN
40	100	750	33348	51448	219360	NRN
40	300	950	11154	56182	220240	NRN
40	400	950	11124	56224	220240	NRN
50	50	1400	59494	119420	600200	NRN
50	100	1600	29404	120771	462900	NRN
50	250	2000	25673	120862	427900	NRN
50	400	1600	16255	120912	302150	NRN
50	400	2000	16255	120912	302150	NRN
50	500	2000	12864	120979	260000	NRN

for Sudoku is given using ACO in the study of Huw Lloyd [16]. Another solution which is the first nature-based algorithm for the NP-Complete Nurikabe problem is presented by Amos et al. [35]. This algorithm was developed based on ACO. For future work, it would be interesting to solve the

Futoshiki problem using Ant Colony Optimization (ACO) and an Artificial Bee Colony (ABC)


algorithm [36]. The performance of these approaches could then be compared with existing solutions, such as the improved constraint programming method developed by Kostyukova and Tchemisova [37].

## References


- [1] Bobga, B., Goldwasser, J.L., Hilton A.J.W. & Johnson P.D. (2011). Completing partial latin squares: Cropper's question. *Australasian Journal of Combinatorics*, 49, 127-152.
- [2] Goldwasser, J., Hilton A., Hoffman D.G. & Ozkan, S. (2015). Hall's theorem and extending partial latinized rectangles. *Journal of Combinatorial Theory Series A*, 130, 26-41. <https://doi.org/10.1016/j.jcta.2014.10.007>
- [3] Euler, R. (2010). On the completability of incomplete Latin squares. *European Journal of Combinatorics*, 31, 535-552. <https://doi.org/10.1016/j.ejc.2009.03.036>
- [4] Colbourn, C.J. (1984). The complexity of completing partial latin squares. *Discrete Applied Mathematics*, 8, 25-30. [https://doi.org/10.1016/0166-218X\(84\)90075-1](https://doi.org/10.1016/0166-218X(84)90075-1)
- [5] Haraguchi, K. & Ono, H. (2014). Approximability of Latin square completion type puzzles. *International Conference on Fun with Algorithms*, 218-229. [https://doi.org/10.1007/978-3-319-07890-8\\_19](https://doi.org/10.1007/978-3-319-07890-8_19)
- [6] Donovan, D. (2010). The completion of partial latin squares. *Australasian Journal of Combinatorics*, 22, 247-264.
- [7] Galvin, F., Stephen, C.L., Kim, S.S. & Callan, D. (2001). A Generalization of Hall's Theorem: 10701. *The American Mathematical Monthly*, 108, 79-80. <https://doi.org/10.2307/2695691>
- [8] Ivanyi, A. & Nemeth, Z. (2011). List coloring of Latin and Sudoku graphs. *8th Joint Conf. on Math. and Comp. Sci.*
- [9] Yato, T. & Setai, T. (2003). Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A, 5, 1052-1060.
- [10] Norvig, P. (2018). Solving every Sudoku puzzle. <http://norvig.com/sudoku.html>.
- [11] Crawford, B., Castro, C. & Monfroy, E. (2009). Solving sudoku with constraint programming. *MCDM, CCIS Communications in Computer and Information Science*, 35, 345-348. [https://doi.org/10.1007/978-3-642-02298-2\\_52](https://doi.org/10.1007/978-3-642-02298-2_52)
- [12] Knuth, D.E. Dancing links. (2000). Millennial Perspectives in Computer Science. *Proceedings of the 1999 Oxford-Microsoft Symposium*, 187-214.
- [13] Glover, F.W. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5), 533-549. [https://doi.org/10.1016/0305-0548\(86\)90048-1](https://doi.org/10.1016/0305-0548(86)90048-1)
- [14] Pacurib, J.A., Seno, G.M.M. & Yusiong, J.P.T. (2009). Solving Sudoku puzzles using improved artificial bee colony algorithm. *In Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*, 885-888. <http://doi.org/10.1109/ICICIC.2009.334>
- [15] Moraglio, A., & Togelius, J. (2007). Geometric particle swarm optimization for the Sudoku puzzle. *In Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, 118-125. <https://doi.org/10.1145/1276958.1276975>
- [16] Lloyd, H. & Amos, M. (2020). Solving sudoku with ant colony optimization, *IEEE Transactions on Games*, 12, 302-311. <https://doi.org/10.1109/TG.2019.2942773>
- [17] Musliu, N., & Winter, F. (2017). A Hybrid Approach for the Sudoku Problem: Using Constraint Programming in Iterated Local Search, *IEEE Intelligent Systems*, 32 (2), 52-62. <https://doi.org/10.1109/MIS.2017.29>
- [18] Lastrina, M.A. (2012). *List-coloring and sum-list coloring problems on graphs*. PhD Thesis, Iawo University.
- [19] Tuza, Z. (1997). Graph colorings with local constraints - a survey. *Discussiones Mathematicae Graph Theory*, 17, 161-228. <https://doi.org/10.7151/dmgt.1049>
- [20] Karp, R.M. (1972). Reducibility among Combinatorial Problems. In: *Complexity of Computer Computations*. The IBM Research Symposia Series, Boston, MA, Springer. [https://doi.org/10.1007/978-1-4684-2001-2\\_9](https://doi.org/10.1007/978-1-4684-2001-2_9)
- [21] Kratochvil, J., & Tuza, Z. (1994). Algorithmic complexity of list coloring. *Discrete Applied Mathematics*, 50(3), 297-302. [https://doi.org/10.1016/0166-218X\(94\)90150-3](https://doi.org/10.1016/0166-218X(94)90150-3)
- [22] Diestel, R. (2017). *Graph Theory*. Graduate Texts in Mathematics, Heidelberg: Springer-Verlag. <https://doi.org/10.1007/978-3-662-53622-3>
- [23] Vizing, V.G. (1976). Coloring the vertices of a graph in prescribed colors. *Diskret. Analiz., Metody Diskret. Anal. v. Teorii Kodov i Shem*, 101, 3-10.
- [24] Lovász, L. (1973). Coverings and coloring of hypergraphs. *Proc. 4th Southeastern Conf. on Combinatorics, Graph Theory and Computing*, 3-12.
- [25] Erdos, P. & Rubin, A.L., Taylor. (1979). Choosability in graphs. *Proceedings of the West Coast Conference on Combinatorics, Graph Theory and Computing*, 26, 125-157.
- [26] Garey, M.R. & Johnson, D.S. *Computers and Intractability, A guide to the theory of NP-Completeness*. W. H. Freeman and Co., 1979.
- [27] Mahmood, A.S. (2019). Design random number generator utilizing the Futoshiki puzzle. *Journal of Information Hiding and Multimedia Signal Processing*, 10, 178-186.

- [28] Haraguchi, K. (2013). The number of inequality signs in the design of Futoshiki puzzle. *Journal of Information Processing*, 21, 26-32. <https://doi.org/10.2197/ipsjjip.21.26>
- [29] Sahu, H.S., Nayak, S.K. & Mishra, S. (2016). Maximizing the power generation of a partially shaded PV array. *IEEE Journal of Emerging and Selected Topics in Power Electronics*, 4, 626-637. <https://doi.org/10.1109/JESTPE.2015.2498282>
- [30] Bondy, J.A. & Murty, U.S.R. (2008). *Graph Theory, Graduate Texts in Mathematics*. Springer, New York. <https://doi.org/10.1007/978-1-84628-970-5>
- [31] Orden, D., Marsa, M.I., Gimenez, G.J.M. & Hoz, E. (2017). Spectrum graph coloring and applications to Wi-Fi channel assignment. *Symmetry*, 10(3), 65. <https://doi.org/10.3390/sym10030065>
- [32] Garg, N., Papatriantafyllou M. & Tsigas, P. (1996). Distributed list coloring: how to dynamically allocate frequencies to mobile base stations. *Eighth IEEE Symposium on Parallel and Distributed Processing*, 18-25. <https://doi.org/10.1109/SPDP.1996.570312>
- [33] Zeitlhofer, T. & Wess, B. (2003). List-coloring of interval graphs with application to register assignment for heterogeneous register-set architectures. *Signal Processing*, 83 (7), 1411-1425. [https://doi.org/10.1016/S0165-1684\(03\)00089-6](https://doi.org/10.1016/S0165-1684(03)00089-6)
- [34] Denes, J. & Keedwell, AD. (1991). *Latin Squares. New Developments in the Theory and Applications*, North Holland.
- [35] Amos, M., Crossley, M. & Lloyd, H. (2019). Solving nurikabe with ant colony optimization. *GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 129-130. <https://doi.org/10.1145/3319619.3338470>
- [36] Bektur, G. & Aslan, H.K. (2024). Artificial bee colony algorithm for operating room scheduling problem with dedicated/flexible resources and cooperative operations. *An International Journal of Optimization and Control: Theories & Applications (IJOCTA)*, 14(3), 193-207. <https://doi.org/10.11121/ijocta.1466>
- [37] Kostyukova, O. & Tchemisova T. (2024). Exploring constraint qualification-free optimality conditions for linear second-order cone programming. *An International Journal of Optimization and Control: Theories & Applications (IJOCTA)*, 14(3), 168-182. <https://doi.org/10.11121/ijocta.1421>

**Banu Baklan Şen** completed her BSc in Mathematics and Computer Science and her MSc in Information Technologies at Bahçeşehir University, Turkey. She defended her PhD in Computer Engineering at Kadir Has University in Turkey. Her academic work includes graph theory algorithms and theoretical computer science. She has participated in international academic collaborations. Her research has led to publications in prominent journals and presentations at international conferences with the expertise in combinatorial optimization and algorithms. She is deeply committed to both teaching and research.

 <https://orcid.org/0000-0003-4545-5044>

**Öznur Yaşar Diner** obtained her BSc in Mathematics from the Middle East Technical University in Turkey. She completed her MSc in Mathematics at the University of Göttingen in Germany and defended her Ph.D. in Mathematics at Memorial University of Newfoundland in Canada. Her primary research interests lie in structural graph theory and theoretical computer science, with a focus on combinatorial problems. She is passionate about teaching and conducting collaborative research.

 <https://orcid.org/0000-0002-9271-2691>

An International Journal of Optimization and Control: Theories & Applications (<http://www.ijocta.org>)



This work is licensed under a Creative Commons Attribution 4.0 International License. The authors retain ownership of the copyright for their article, but they allow anyone to download, reuse, reprint, modify, distribute, and/or copy articles in IJOCTA, so long as the original authors and source are credited. To see the complete license contents, please visit <http://creativecommons.org/licenses/by/4.0/>.