A Fast Local Search Algorithm for the Latin Square Completion Problem

Shiwei Pan, Yiyuan Wang*, Minghao Yin*

School of Computer Science and Information Technology, Northeast Normal University, China
Key Laboratory of Applied Statistics of MOE, Northeast Normal University, Changchun, China
pansw779@nenu.edu.cn, yiyuanwangjlu@126.com, ymh@nenu.edu.cn

Abstract

The Latin square completion (LSC) problem is an important NP-complete problem with numerous applications. Given its theoretical and practical importance, several algorithms are designed for solving the LSC problem. In this work, to further improve the performance, a fast local search algorithm is developed based on three main ideas. Firstly, a reduction reasoning technique is used to reduce the scale of search space. Secondly, we propose a novel conflict value selection heuristic, which considers the history conflicting information of vertices as a selection criterion when more than one vertex have equal values on the primary scoring function. Thirdly, during the search phase, we record previous history search information and then make use of these information to restart the candidate solution. Experimental results show that our proposed algorithm significantly outperforms the state-of-the-art heuristic algorithms on almost all instances in terms of success rate and run time.

Introduction

A Latin square of order n is an array of n symbols (i.e., $\{1, 2, \ldots, n\}$) in which each symbol occurs exactly once in each row and exactly once in each column. If some grids are empty, then the Latin square complete (LSC) problem of order n aims to complete the empty grids with n symbols to obtain an arbitrary legal Latin square. In the past decades, the LSC problem has been already used in various fields (Laywine and Mullen 1998; Lakić 2001; Gogate and Dechter 2011), such as optical networks (Kumar, Russell, and Sundaram 1999), error correcting codes (Colbourn, Klove, and Ling 2004) and combinatorial designs (Colbourn 2010). Also, the LSC problem can be modeled as the formulas of Boolean satisfiability (SAT) using the extended encoding proposed by (Gomes and Shmoys 2002).

As is known, the LSC problem has been shown to be an NP-complete problem (Colbourn 1984). For the optimized version of the LSC problem, i.e., the partial Latin square extension (PLSE) problem, researchers have designed many approximation algorithms. For example, two classical approximation algorithms were proposed with non-trivial worst-case performance guarantees (Kumar, Russell, and Sundaram 1999). Afterwards, an $e/(e-1)$–approximation algorithm was presented based on the linear programming relaxation of a packing linear programming formulation (Gomes, Regis, and Shmoys 2004). Hajirasouliha et al. (2007) introduced a $(2/3-\varepsilon)$–approximation algorithm for the PLSE problem. It is common to see that approximation algorithms usually have poor performance in practice. There are mainly two types of algorithms for the LSC problem, i.e., exact algorithms and heuristic algorithms.

In the recent decade, several exact algorithms have been proposed for solving the LSC problem. Gomes and Shmoys (2002) proposed three exact algorithms for solving the LSC problem, including a constraint satisfaction problem (CSP) based algorithm, a hybrid algorithm based on linear programming and CSP as well as a SAT-based algorithm. In this work, a common feature of these proposed search algorithms is the careful use of randomization and restarts to obtain some robust solvers, while maintaining the completeness of backtrack search approaches. A systematic comparison of SAT and CSP models was proposed for the LSC problem (Ansótegui et al. 2004). Results show that the above exact algorithms can only solve instances with small sizes.

For solving instances with large sizes, some heuristic LSC algorithms have been proposed, which can obtain an approximate solution within reasonable time. Representative heuristic algorithms for the LSC problem mainly used some neighborhood search techniques. For example, Haraguchi (2015; 2016) designed three efficient neighborhood search algorithms called 1-ILS*, 2-ILS and 3-ILS. Besides, the author also proposed a novel swap operation called Trellis-swap, resulting in a novel Trellis-neighborhood search algorithm named Tr-ILS*. According to the literature, the current best heuristic algorithm for the LSC problem is called MMCOL (Jin and Hao 2019), which is mainly based on a constraint propagation technique, a problem-specific crossover operator, an iterated tabu search procedure and a distance-quality-based pool updating strategy. Besides, a transformation method is also proposed to convert an LSC instance to a domain-constrained Latin square graph (Jin and Hao 2019). Although the MMCOL algorithm performs very well on some hard graphs, it has to cost lots of computation time for obtaining an arbitrary legal solution.

Although previous works made progress in solving the

*Corresponding author

LSC problem in terms of success rate, the performance is still not satisfactory since they usually wasted lots of computation time. For many LSC applications (Barry and Humblet 1993; Ansótegui et al. 2004), the resource of computation time is very important. To address this, we use the Latin square graph to denote the LSC problem and then develop a fast local search algorithm based on three main novel ideas.

Firstly, we design a reduction reasoning-based initialization method called Construct for constructing an initial solution, which can be divided into two phases: reduction and construction. In the reduction phase, three reduction rules are used to fix several grids with specific symbols, which can reduce the scale of problems. In the construction phase, Construct utilizes a simple and fast construction process to generate a solution served as the subsequent search process.

Secondly, a conflict value selection heuristic is presented to decide which moving operation should be selected to update the candidate solution. In the proposed selection heuristic, we first use a very common function as the primary scoring function, which can intuitively reflect the changes of the solution quality regarding to the candidate solution. To deal with the issue about tie-breaking in the primary scoring function, we propose a novel function as the secondary scoring function. We quantify the history conflicting information of each vertex over a period of time, denoted as $cscore$. Our secondary scoring function is based on the $cscore$ values, which measures the neighborhood changing information of candidate solution during the search history.

Thirdly, we propose the history information perturbation mechanism to restart the local search process, which includes the pool updating and solution perturbing. In the pool updating, we employ a solution pool to store the best solutions. To maintain and update the solution pool, two key concepts are defined, including the property of vertex ($state$) and the definition of similarity ($\approx$). In the solution perturbing, we refer to the $state$ values of vertices as the selection criterion to modify the candidate solution.

By incorporating these three ideas, we develop a local search algorithm for the LSC problem called FastLSC. Extensive experiments are carried out to evaluate FastLSC on two classical benchmarks used in previous literature. Experimental results show that FastLSC outperforms its competitors on almost all instances in terms of success rate and run time. Besides, we also conduct experiments to analyze the effectiveness of the proposed ideas.

The remainder of the paper is organized as follows. The next section introduces some basic definitions. Section 3 presents a reduction reasoning-based initialization method. Section 4 presents a conflict value selection heuristic designed for the LSC problem. Section 5 describes our FastLSC algorithm based on the history information perturbation mechanism. Experimental results are shown in Section 6 and Section 7 gives concluding remarks.

Preliminaries

An arbitrary legal Latin square $\mathcal{L}^n$ is an $n \times n$ array filled with n different symbols, each occurring exactly once in each row and exactly once in each column. If some grids are empty, then $\mathcal{L}_p^n$ is called a partial Latin square. The Latin square completion (LSC) problem aims to fill symbols (i.e., $\{1, 2, \ldots, n\}$) to empty grids of $\mathcal{L}_p^n$ to obtain an arbitrary legal Latin square.

Review of Latin Square Graph

Recently, Jin and Hao (2019) define a Latin square graph $G = (V, E)$ to intuitively show a partial Latin square $\mathcal{L}_p^n$, where $V = \{v_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq n\}$ represents all grids and vertex v_{ij} denotes a grid on the ith row and the jth column. If two grids $u, w \in V$ are in the same row or column, then we say $(u, w) \in E$. Thus, $|V| = n^2$ and $|E| = n^2(n-1)$. For each vertex $v \in V$, the neighbors of vertex v is defined as $N(v) = \{u \in V \mid (v, u) \in E\}$, and the degree of vertex is $d(v) = |N(v)| = 2(n-1)$. For $\forall v_{ij} \in V$, its row vertex set is denoted as $RN(v_{ij}) = \{v_{ik} \mid 1 \leq k \leq n, k \neq j\}$, while its column vertex set is denoted as $CN(v_{ij}) = \{v_{kj} \mid 1 \leq k \leq n, k \neq i\}$. We can easily get that $N(v) = RN(v) \cup CN(v)$.

An independent set I is a subset of V such that no two vertices are adjacent, i.e., $(v, u) \notin E$ for $\forall v, u \in I$. A legal n-coloring is a partition of V into n independent sets (color classes), i.e., $\mathcal{V}_l^n = \{V_i \mid 1 \leq i \leq n\}$. For a Latin square graph $G = (V, E)$, the LSC problem can be encoded as the precoloring extension problem (PEP) (Biro, Hujter, and Tuza 1992), and the size of color classes is n. In the PEP problem, we use $D(v) = \{V_j\}$ to denote that vertex v has been already put into a fixed color class V_j, while the *color domain* of each remaining vertex u is denoted as $D(u) = \{V_1, \ldots, V_n\}$. Thus, the aim of the LSC problem can be seen as finding a legal n-coloring where the number of vertices for each color class is exactly n. During the search process, $\mathcal{V}^n$ is used to denote the current set of color classes.

A Novel Reduction Reasoning-Based Initialization Method

In this section, we first design three reduction rules to simplify the problem instances and then introduce the initialization process under the reduced instances.

Reduction Rules

To improve the performance of local search for the LSC problem on the hard instances with large sizes, three reduction rules are introduced in which the first reduction rule has been already used into reducing the scale of LSC instances (Jin and Hao 2019).

Reduction Rule 1: If a vertex v has only one optional color class (i.e., $D(v) = \{V_i\}$), then vertex v should be put into the color class V_i, $D(v) = \emptyset$ and $D(u) = D(u) \setminus \{V_i\}$ for $\forall u \in N(v)$.

Reduction Rule 2: For a vertex v, $S_v^r = D(v) \setminus \cup_{u \in RN(v)} D(u)$. If the size of S_v^r is exactly one (i.e., $S_v^r = \{V_i\}$), then vertex v should be put into the only one color class V_i, $D(v) = \emptyset$ and $D(u) = D(u) \setminus \{V_i\}$ for $\forall u \in N(v)$.

Reduction Rule 3: For a vertex v, $S_v^c = D(v) \setminus \cup_{u \in CN(v)} D(u)$. If the size of S_v^c is exactly one (i.e., $S_v^c = \{V_i\}$), then vertex v should be put into the only one color class V_i, $D(v) = \emptyset$ and $D(u) = D(u) \setminus \{V_i\}$ for $\forall u \in N(v)$.

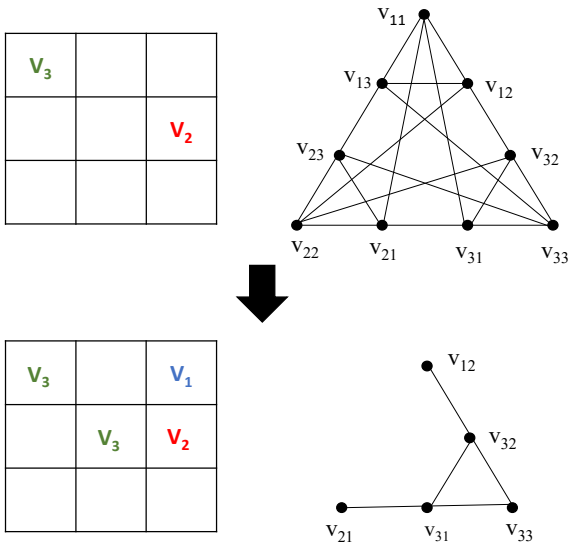

Figure 1: An example of three reduction rules.

Algorithm 1: $Construct(G)$

Input: A Latin square graph $G = (V, E)$ where $|V| = n^2$
Output: The set of color classes $\mathcal{V}^n$

1 initialize a color domain $D(v)$ for $\forall v \in V$;
2 initialize a color class V_i for $\forall V_i \in \mathcal{V}^n$;
3 $CandSet := V$;
4 **while** *$CandSet$ is not empty* **do**
5 **if** *there exists vertex $v_1 \in CandSet$ satisfying any reduction rule or as a filed grid* **then**
6 $V := V \setminus \{v_1\}$;
7 put vertex v_1 into a color class V_i based on reduction rule;
8 $CandSet := CandSet \setminus \{v_1\}$;
9 $D(u) := D(u) \setminus \{V_i\}$ for $\forall u \in N(v_1)$;
10 **else break** ;
11 **while** *$CandSet$ is not empty* **do**
12 select a random vertex v_2 from $CandSet$;
13 $CandSet := CandSet \setminus \{v_2\}$;
14 put vertex v_2 into a random color class selected from $D(v_2)$;
15 **return** $\mathcal{V}^n$;

The correctness of the second and third reduction rules is very obvious. Each grid in the same row (or column) needs to be filled with different symbols. Assuming that the color domain of vertex v contains one special color class V_i which doesn't occur in the color domain of any vertex in v's row (or column) vertex set. It means that vertex v has only one option that v should be put into V_i. Note that, as far as we know, although these two rules are relatively simple, they have never been applied into heuristic LSC algorithms.

To make readers clearly understand our reduction rules, we present an example in Figure 1, including a partial Latin square $\mathcal{L}_p^3$ with 2 filled grids and 7 empty grids as well as the corresponding Latin square graph with 9 vertices and 18 edges. Assuming that the set of color classes $\mathcal{V}^3 = \{V_1, V_2, V_3\}$. Firstly, we can put v_{11} and v_{23} into V_3 and V_2 since these two grids are filled ones. Afterwards, v_{13} is reduced by filling it with V_1 according to the reduction rule 1. The next reduction vertex is v_{22} based on the reduction rule 2. Note that when a vertex v is reduced, we will also update the corresponding $D(u)$ for $\forall u \in N(v)$. In this example, we can reduce all vertices based on our reduction rules, and then obtain a legal Latin square.

The Novel Initialization Algorithm

By utilizing the proposed reduction rules, we can iteratively fix a considerable portion of the grids during the initial reduction process. Afterwards, we construct an initial LSC solution by using a simple and fast construction process. Based on the reduction and construction processes, we propose a novel reduction reasoning-based initialization algorithm in Algorithm 1.

At the beginning, the algorithm initials the color domain of each vertex, which is mainly divided into two kinds of vertices, i.e., filled and empty grids. For each filled grid v_{ij}, its color domain $D(v_{ij})$ is set to its fixed color class V_x, while for each remaining empty grid v'_{ij}, its color domain $D(v'_{ij})$ is set to $\{V_1, \ldots, V_n\}$. Each color class V_i^n needs to be initialized (line 2). In the line 3, the candidate vertex set $CandSet$ is set to V. Afterwards, the algorithm comes into the initial reduction process (lines 4–10). If the algorithm can remove a vertex v_1 based on the reduction rules, then the algorithm will remove it and then fix its color set (lines 5–7). The corresponding $CandSet$ and the color domain of v's neighbors should be updated (lines 8 and 9).

After any vertices cannot be removed, the algorithm turns to the simple and fast construction process (lines 11-14). During the construction process, the algorithm first selects a random vertex v_2 and removes it from $CandSet$ (lines 12 and 13). The algorithms picks a random color class from the color domain of v_2 and then adds v_2 into this color class (line 14). At last, $\mathcal{V}^n$ is returned (line 15).

The Conflict Value Selection Heuristic

We design a selection heuristic based on the definition of conflict value. Before proposing this heuristic, we first introduce the primary scoring function.

The Primary Scoring Function

To define the primary scoring function, we first give some necessary concepts. Assuming that the current color class set is $\mathcal{V}^n$ and $V_k \in \mathcal{V}^n$. If $v, u \in V_k$ and $(v, u) \in E$, then (v, u) is called a *conflict edge*. We use $CL(\mathcal{V}^n)$ to denote the total number of conflict edges in $\mathcal{V}^n$. Moving a vertex v from a color class V_i to another color class V_j is denoted as $Move(v, V_i, V_j)$, which leads to a neighboring color class set $\mathcal{V}_c^n$. When performing an operation $Move(v, V_i, V_j)$, we formally define the primary scoring function as below.

$$pscore(v, V_i, V_j) = CL(\mathcal{V}^n) - CL(\mathcal{V}_c^n)$$

Note that the primary scoring function $pscore$ intuitively reflects the effects of the moving operation for the current color class set.

The Secondary Scoring Function

The primary scoring function always fails to select the sole moving operation. Thus, to further select a moving operation among these operations with the same best $pscore$, we design the secondary scoring function based on the definition of conflict value.

We first define the conflict value of vertex v, denoted as $cscore(v)$. Initially, the $cscore$ value of each vertex is set to 0. Two updating rules are proposed to maintain the $cscore$ value of each vertex as follow.

Updating Rule 1: After each iteration of local search, $cscore(v)$ and $cscore(u)$ both are increased by one, for each conflict edge (v, u).

Updating Rule 2: When performing $Move(v, V_i, V_j)$, $cscore(v)$ is reset to 0. If there exists vertex $u \in V_i$ and $N(u) \cap V_i = \{v\}$, then $cscore(u) = 0$.

In the second rule, if moving v will eliminate all conflict edges about u, then $cscore(u)$ is also reset to 0.

Intuitions underlying the $cscore$ are given below. $cscore$ reflects either how long a vertex v stays in its color class since the last time it was moved or that the number of v's conflict edges becomes 0 due to moving another vertex's position. Furthermore, $cscore$ accumulates the number of the conflict edges over this period of time. Thus, using the $cscore$ values makes candidate vertices be assigned to different selection priorities based on the search information.

Selection Rule

For any vertex v, the $pscore(v, V_i, V_j)$ value can be considered as the immediate impact value, while $cscore$ reflects the history conflicting information of vertices over a period of time. Combining the above two scoring functions, our novel selection rule is proposed.

Selection Rule: pick an operation $Move(v, V_i, V_j)$ with the biggest $pscore$ value, breaking ties by preferring the one with the biggest $cscore$ value, further ties are broken randomly.

The intuition behind our selection rule is to choose the move that immediately reduces the most the conflicts and tie breaking by choosing the vertex that has not been impacted (directly or indirectly) by a move for the longest time.

The FastLSC Algorithm

Based on the novel initialization method and the conflict value selection heuristic, we develop a local search algorithm for the LSC problem named FastLSC. To deal with the cycling problem, we use the same tabu strategy (Glover and Laguna 1998; Jin and Hao 2019), i.e., recording the moving operation (v, V_i, V_j) to prevent putting a just moved vertex v back into V_i for the next β iterations.

We first introduce some notations. Parameter α denotes the search depth. $\mathcal{V}_c^n$ and $\mathcal{V}_b^n$ are used to denote the current set and the best obtained set of color classes, respectively. We use v^b and v^c to denote vertex v in the respective sets $\mathcal{V}_b^n$ and $\mathcal{V}_c^n$. $iter$ is the current step during the search process, while we use $Pool$ to denote the solution pool, which is used to store the best solutions.

Algorithm 2: the FastLSC algorithm

Input: A Latin square graph $G = (V, E)$ where $|V| = n^2$, the $cutoff$ time
Output: The set of color classes $\mathcal{V}_b^n$

1 $\mathcal{V}_c^n := \mathcal{V}_b^n := Construct(G)$ and $iter := 0$;
2 initialize the solution set $Pool$;
3 $v^c.state := 0$ for $\forall v^c \in V$;
4 **while** $elapse\ time < cutoff$ **do**
5 $depth := 0$;
6 $cscore(v^c) := 0$, for $\forall v^c \in V$;
7 **while** $depth < \alpha$ **do**
8 **if** $CL(\mathcal{V}_c^n) \leq CL(\mathcal{V}_b^n)$ **then**
9 $\mathcal{V}_b^n := \mathcal{V}_c^n$;
10 $v^b.state := v^c.state$ for $\forall v \in V$;
11 select a moving operation $Move(v^c, V_i^c, V_j^c)$ based on selection rule and tabu strategy;
12 $V_i^c := V_i^c \setminus \{v^c\}$ and $V_j^c := V_j^c \cup \{v^c\}$;
13 update the corresponding $cscore$ values based on two updating rules;
14 $depth := depth + 1$ and $iter := iter + 1$;
15 $v^c.state := iter$;
16 **if** $CL(\mathcal{V}_b^n) == 0$ **then return** $\mathcal{V}_b^n$;
17 $\mathcal{V}_c^n := Perturb(\mathcal{V}_b^n, Pool)$;
18 **return** $\emptyset$;

In our FastLSC algorithm, for storing the information of vertex $v \in V$, we define an additional property: state, denoted by $v.state$. In the beginning, for each vertex $v \in V$, the $v.state$ is set to 0. Then, whenever the v is moved from one color class to another one, $v.state$ is set to the number of current step (i.e., $iter$). The pseudo code of FastLSC is shown in Algorithm 2.

Now we describe the FastLSC algorithm in detail. At the beginning, $\mathcal{V}_c^n$ and $\mathcal{V}_b^n$ are generated by calling $Construct(G)$ (line 1). During this process, some redundant vertices will be removed. The value of $iter$, the set of solutions $Pool$ and the $step$ value of each vertex should be initialized accordingly. There is an outer loop (lines 4–17) and an inner loop (lines 7–16). In each inner loop ($depth < \alpha$), the algorithm searches for the set of color classes with the smaller total number of conflict edges. Before each inner loop, $depth$ needs to be reset to 0 (line 5) and the algorithm initializes the $cscore$ of each vertex (line 6). After each inner loop, the algorithm uses a novel perturbation method ($Perturb$) to obtain the initial solution for the next round, which will be introduced in the next subsection (line 17). Finally, if the algorithm fails to find any arbitrary legal solution, then the algorithm returns $\emptyset$ when a $cutoff$ time is reached (line 18).

In each iteration of the inner loop, FastLSC chooses one moving operation to modify the current set of color classes $\mathcal{V}_c^n$. First, if the total number of conflict edges for $\mathcal{V}_c^n$ is not bigger than that for $\mathcal{V}_b^n$, then $\mathcal{V}_b^n$ and the related state information will be updated accordingly (lines 8–10). Afterwards, the algorithm turns to select a moving operation via using our proposed selection rule and tabu strategy, and then moves vertex v from V_i^c to V_j^c. After then, the correspond-

Algorithm 3: The $Perturb$ Function

Input: The best obtained set of color classes $\mathcal{V}_b^n$ and the solution set $Pool$
Output: Initial solution $\mathcal{V}_c^n$

1 **if** $CL(\mathcal{V}_b^n) < CL(\mathcal{V}_j^n)$ *for* $\forall CL(\mathcal{V}_j^n) \in Pool$ **then**
2 remove all solutions from $Pool$;
3 $Pool := Pool \cup \{\mathcal{V}_b^n\}$ and $\mathcal{V}_c^n := \mathcal{V}_b^n$;
4 **else**
5 **if** $\mathcal{V}_i^n \approx \mathcal{V}_b^n$ *for* $\exists \mathcal{V}_i^n \in Pool$ **then**
6 $v_j^i.state := v_j^b.state$ for $\forall v_j \in V$;
7 select a random one $\mathcal{V}_c^n$ from $Pool$;
8 **else**
9 **if** $|Pool| < pool\_size$ **then**
10 $Pool := Pool \cup \{\mathcal{V}_b^n\}$
11 **else**
12 remove the oldest one $\mathcal{V}_j^n$ from $Pool$;
13 $Pool := Pool \cup \{\mathcal{V}_b^n\}$;
14 $\mathcal{V}_c^n := \mathcal{V}_b^n$;
15 $minIter := min\{v_j^c.state \mid v_j^c \in V\}$;
16 $maxIter := max\{v_j^c.state \mid v_j^c \in V\}$;
17 $RCL := (maxIter - minIter) \times \theta + minIter$;
18 $C := \emptyset$;
19 **for** $\forall v_j^c \in V$ **do**
20 **if** $v_j^c.state \leq RCL$ **then** $C := C \cup \{v_j^c\}$;
21 $cnt := \lfloor |C|/2 \rfloor$;
22 **while** $cnt > 0$ **do**
23 pop a random vertex v^c from C;
24 select a random color class V_i^c from $D(v^c)$;
25 put v^c into the V_i^c of $\mathcal{V}_c^n$;
26 $cnt := cnt - 1$;
27 **return** $\mathcal{V}_c^n$;

ing $cscore$, $depth$ and $iter$ should be updated (lines 13 and 14). In the next step, the algorithm uses $v^c.state$ to record the current number of steps (line 15). At last, if $\mathcal{V}_b^n$ is an arbitrary legal set of color classes, then the algorithm will return it (line 16).

The Perturb Function

Local search algorithms usually use some perturbation methods to diversify the solution when meeting the local optimal (Cai, Luo, and Zhang 2017; Xu, He, and Li 2019; Wang et al. 2020a,b). In the solution restart phase of our algorithm, an important component is called history information perturbation mechanism ($Perturb$), which restarts the algorithm by constructing a new solution based on previous history information when falling into the local optima. Specially, we filter the optimal solution obtained each time and add it to the solution pool if possible. Afterwards, we will select a solution from the solution pool, and then modify this solution by some perturbation strategies as the initial solution for the next round of local search. As we know, our perturbation mechanism is a novel idea by combining the respective advantage of population-based search and powerful local search.

Before introducing the $Perturb$ function, we first define the similarity ($\approx$) of two solutions as below.

Definition 1. *For two solutions $\mathcal{V}_i^n$ and $\mathcal{V}_j^n$, if these two solutions have the same conflict edges and each conflict edge on these solutions belongs to the same color class respectively, then we call $\mathcal{V}_i^n$ and $\mathcal{V}_j^n$ are similarity, denoted as $\mathcal{V}_i^n \approx \mathcal{V}_j^n$.*

Here, we give an example to explain the definition of similarity. Suppose that $\mathcal{V}_1^3 = \{\{v_{11}, v_{22}, v_{33}\}, \{v_{12}, v_{13}, v_{21}, v_{31}\}, \{v_{23}, v_{32}\}\}$ and $\mathcal{V}_2^3 = \{\{v_{11}, v_{23}, v_{32}\}, \{v_{12}, v_{13}, v_{21}, v_{31}\}, \{v_{22}, v_{33}\}\}$. Obviously, $\mathcal{V}_1^3$ and $\mathcal{V}_2^3$ are not the same, but they have the same conflict edges. All conflict edges belong to the same second color class. Thus, $\mathcal{V}_1^3 \approx \mathcal{V}_2^3$.

The pseudo code of $Perturb$ is introduced in Algorithm 3. The $Perturb$ function mainly includes two phases: the updating phase of the solution pool (lines 1–14) and the reconstruction solution phase based on the information of vertex state (lines 15–26).

In the first phase, if $\mathcal{V}_b^n$ is better than any solution of $Pool$, then the algorithm clears all solutions from $Pool$ and then adds the best solution $\mathcal{V}_b^n$ into $Pool$ (lines 1–3). $\mathcal{V}_b^n$ will be used as the initial perturbation solution. Otherwise, the algorithm determines whether $\mathcal{V}_b^n$ is similar to a solution $\mathcal{V}_i^n$ in the $Pool$. If so, then the state value of each vertex in $\mathcal{V}_i^n$ should be updated by $\mathcal{V}_b^n$ (line 6). Then, we choose a random solution $\mathcal{V}_c^n$ from the $Pool$ as the initial perturbation solution (line 7). If there are no such similar solutions in $Pool$, then the algorithm needs to add $\mathcal{V}_b^n$ into the $Pool$. If the number of solutions in the $Pool$ is less than $pool\_size$, then the algorithm will directly add $\mathcal{V}_b^n$ into the $Pool$ (line 10). In our work, $pool\_size$ is set to 20, i.e., storing at most 20 solutions. Otherwise, we will replace the oldest solution in the solution pool $Pool$ with $\mathcal{V}_b^n$ (lines 11–13). The oldest solution means that this solution stays the longest time in the solution pool. In the following step, $\mathcal{V}_b^n$ is selected as the initial perturbation solution (line 14).

In the second phase, the restrict candidate list RCL is computed based on the maximum and minimum values of $state$ (lines 15–17). Then, RCL is used to pick some vertices whose $state$ value is smaller than RCL and then the algorithm adds these satisfied vertices into C (lines 18–20). In the next steps, we randomly select half of vertices in C. For each selected vertex, the algorithm will randomly change the color classes of these vertices based on their respective color domains (lines 24 and 25).

Experimental Results

We carry out extensive experiments to evaluate the performance of FastLSC. We compare FastLSC with five state-of-the-art heuristic algorithms: 1-ILS* (2016), 2-ILS (2016), 3-ILS (2016), Tr-ILS* (2016) and MMCOL (2019). The code of MMCOL was kindly provided by the authors[1]. Because the remaining four algorithms are not available to us, we have to compare their results in the literature by using the same $cutoff$ time. All algorithms are implemented in C++ and compiled by g++ with '-O3' option. All experiments of

[1] http://www.info.univ-angers.fr/~hao/lsc.html

our algorithm and competitors are run on Intel Xeon E5-2640 v4 @ 2.40GHz CPU with 128GB RAM under CentOS 7.5.

For our experiments, we considered two popular benchmarks including the random benchmark [2] and the COLOR03 benchmark[3], which has already used into testing the performance of previous heuristic LSC algorithms (Gomes and Shmoys 2002; Haraguchi 2016; Jin and Hao 2019). The random benchmark consists of 1800 instances. There are 18 families, each of which contains 100 instances with the same type. For each family, QWH-n-r denotes that n is the order of Latin square and r is the ratio of filled grids over the $n \times n$ grids. As for the COLOR03 benchmark, we use the same 19 traditional instances as previous works.

For a competitor MMCOL, we set the parameters as same as what described in the corresponding literature. There are three parameters in our algorithm. For the sake of fairness, the search depth α and the tabu strength β use the same values as MMCOL, i.e., $\alpha = 10^5$ and $\beta = rand()\%10 + 0.6 \times CL(\mathcal{V}^n)$. For our other parameter, we set θ to 0.2 according to our preliminary experiment.

For each instance, all algorithms are executed 30 times with random seeds 1,2,3. . . 30. Each time terminates upon either finding an arbitrary legal solution or reaching a given $cutoff$ time denoted as ct. For all algorithms, we test them under three $cutoff$ time, 10, 100 and 1000 seconds. For each algorithm, we report the number of successful runs $\#suc$. For all random seeds, $time$ (in seconds) denotes the mean value of the run time when an algorithm obtains an arbitrary legal solution. For the random benchmark, we report for each family the averaged value of $time$, denoted as $\overline{time}$. Besides, for each family in the random benchmark, we use suc_t to denote the number of instances for which an algorithm can obtain at least one arbitrary legal Latin square for the same instance over 30 times. The bold values in the tables indicate the best solution among all the algorithms.

Results on Random Benchmark

Because we failed to obtain the source codes of four ILS versions, we directly used their experimental results where the $cutoff$ time was set to 10 seconds in the literature. We run FastLSC and MMCOL under the same $cutoff$ time. Table 1 presents that the best ILS algorithm Tr-ILS* can only get all results for 6 families, while FastLSC and MMCOL both can obtain all results for 15 families under the same $cutoff$ time. Observed from the results of Table 1, the performance of FastLSC and MMCOL totally dominates the remaining four competitors. Thus, in the following part, we mainly compare FastLSC with MMCOL.

Table 2 shows the comparison results between FastLSC and MMCOL under different $cutoff$ time. For three instance families (i.e., $r = 70$), FastLSC shows superiority to MMCOL in terms of both success rate and run time, while the success rates of FastLSC and MMCOL on the remaining families are always 100% under different $cutoff$ time. To further verify the performance of FastLSC, Table 3 displays the detailed results where either FastLSC or MMCOL fails to obtain 100% success rate under 30 times on 1800 random instances. Results show that FastLSC performs better than MMCOL for almost all instances with the exception of three cases (i.e., QWH-50-70-49, QWH-50-70-51 and QWH-60-70-99). Particularly, FastLSC cannot achieve 100% success rate for only 15 instances, while MMCOL fails to reach 100% success rate for 74 instances.

[2]https://github.com/YanJINFR/Latin-Square-Completion.git

[3]http://mat.gsia.cmu.edu/COLOR03/

Instance Family	1-ILS* ct=10s	2-ILS	3-ILS	Tr-ILS*	MMCOL	FastLSC
	suc_t	suc_t	suc_t	suc_t	suc_t	suc_t
50-30	**100**	**100**	95	**100**	**100**	**100**
50-40	99	99	92	**100**	**100**	**100**
50-50	96	96	83	**100**	**100**	**100**
50-60	30	23	5	36	**100**	**100**
50-70	0	0	0	0	28	**99**
50-80	**100**	**100**	**100**	**100**	**100**	**100**
60-30	**100**	**100**	51	**100**	**100**	**100**
60-40	96	99	52	**100**	**100**	**100**
60-50	89	95	17	95	**100**	**100**
60-60	16	12	0	23	**100**	**100**
60-70	0	0	0	0	8	**94**
60-80	98	**100**	99	99	**100**	**100**
70-30	**100**	**100**	19	99	**100**	**100**
70-40	95	97	8	98	**100**	**100**
70-50	82	87	0	84	**100**	**100**
70-60	5	2	0	10	**100**	**100**
70-70	0	0	0	0	0	**82**
70-80	93	97	95	98	**100**	**100**

Table 1: Results of FastLSC and all competitors in the random benchmark. We use n-r to denote QWH-n-r.

Instance Family	MMCOL ct=1000s		FastLSC		MMCOL ct=100s		FastLSC	
	$\#suc$	$\overline{time}$	$\#suc$	$\overline{time}$	$\#suc$	$\overline{time}$	$\#suc$	$\overline{time}$
50-30	**3000**	0.15	**3000**	**0.11**	**3000**	0.15	**3000**	**0.11**
50-40	**3000**	0.12	**3000**	**0.09**	**3000**	0.12	**3000**	**0.09**
50-50	**3000**	0.15	**3000**	**0.11**	**3000**	0.15	**3000**	**0.11**
50-60	**3000**	1.32	**3000**	**0.72**	**3000**	1.32	**3000**	**0.72**
50-70	2814	221.91	**2919**	70.63	694	63.77	**2417**	26.66
50-80	**3000**	**<0.01**	**3000**	**<0.01**	**3000**	**<0.01**	**3000**	**<0.01**
60-30	**3000**	0.34	**3000**	**0.26**	**3000**	0.34	**3000**	**0.26**
60-40	**3000**	0.27	**3000**	**0.20**	**3000**	0.27	**3000**	**0.20**
60-50	**3000**	0.34	**3000**	**0.26**	**3000**	0.34	**3000**	**0.26**
60-60	**3000**	3.30	**3000**	**1.69**	**3000**	3.30	**3000**	**1.69**
60-70	2971	230.43	**2999**	40.99	246	80.09	**2776**	30.17
60-80	**3000**	**<0.01**	**3000**	**<0.01**	**3000**	**<0.01**	**3000**	**<0.01**
70-30	**3000**	0.69	**3000**	**0.52**	**3000**	0.69	**3000**	**0.52**
70-40	**3000**	0.55	**3000**	**0.40**	**3000**	0.55	**3000**	**0.40**
70-50	**3000**	0.74	**3000**	**0.50**	**3000**	0.74	**3000**	**0.50**
70-60	**3000**	6.81	**3000**	**3.05**	**3000**	6.81	**3000**	**3.05**
70-70	2918	365.98	**3000**	45.39	9	68.76	**2773**	35.41
70-80	**3000**	0.04	**3000**	**<0.01**	**3000**	0.04	**3000**	**<0.01**

Table 2: Results of MMCOL and FastLSC in the random benchmark under different $cutoff$ time. We use n-r to denote QWH-n-r.

Instance	MMCOL		FastLSC		Instance	MMCOL		FastLSC	
	$\#suc$	$\overline{time}$	$\#suc$	$\overline{time}$		$\#suc$	$\overline{time}$	$\#suc$	$\overline{time}$
QWH-50-70-100	18	603.74	**26**	331.17	QWH-70-70-1	29	374.25	**30**	30.02
QWH-50-70-13	16	465.31	**26**	295.16	QWH-70-70-100	29	322.08	**30**	52.83
QWH-50-70-20	10	702.51	**14**	416.95	QWH-70-70-15	29	287.84	**30**	38.08
QWH-50-70-21	24	350.01	**30**	262.17	QWH-70-70-18	27	462.08	**30**	41.04
QWH-50-70-25	27	296.44	**30**	130.15	QWH-70-70-20	29	320.54	**30**	35.63
QWH-50-70-26	5	681.32	**15**	394.72	QWH-70-70-22	29	358.13	**30**	39.63
QWH-50-70-27	28	345.83	**30**	100.16	QWH-70-70-23	26	511.47	**30**	44.08
QWH-50-70-28	**29**	449.32	**29**	**143.12**	QWH-70-70-24	29	424.61	**30**	39.40
QWH-50-70-29	29	366.17	**30**	68.84	QWH-70-70-25	23	527.26	**30**	93.41
QWH-50-70-35	27	386.54	**30**	75.67	QWH-70-70-26	28	469.08	**30**	57.65
QWH-50-70-36	29	446.89	**30**	110.86	QWH-70-70-28	29	426.80	**30**	36.70
QWH-50-70-38	29	326.07	**30**	77.28	QWH-70-70-29	28	469.39	**30**	66.30
QWH-50-70-39	29	300.28	**30**	79.73	QWH-70-70-30	22	551.02	**30**	72.23
QWH-50-70-40	26	428.58	**30**	75.53	QWH-70-70-32	26	606.84	**30**	54.97
QWH-50-70-45	29	298.12	**30**	45.73	QWH-70-70-36	26	553.23	**30**	56.11
QWH-50-70-47	29	188.02	**30**	41.02	QWH-70-70-37	29	411.17	**30**	44.47
QWH-50-70-49	**30**	151.14	29	16.67	QWH-70-70-4	27	425.47	**30**	42.28
QWH-50-70-51	**30**	380.85	29	101.96	QWH-70-70-40	29	471.37	**30**	42.77
QWH-50-70-57	2	422.46	**3**	920.68	QWH-70-70-41	27	491.50	**30**	66.61
QWH-50-70-58	15	541.68	**28**	314.77	QWH-70-70-43	29	409.64	**30**	48.47
QWH-50-70-59	29	307.05	**30**	33.89	QWH-70-70-45	27	673.50	**30**	60.43
QWH-50-70-6	25	420.07	**30**	177.28	QWH-70-70-47	27	561.32	**30**	44.41
QWH-50-70-65	**29**	328.80	**29**	**69.78**	QWH-70-70-48	27	455.67	**30**	39.54
QWH-50-70-70	23	358.27	**30**	200.98	QWH-70-70-50	29	460.45	**30**	47.26
QWH-50-70-72	27	321.14	**29**	122.05	QWH-70-70-53	29	283.05	**30**	30.70
QWH-50-70-74	12	525.87	**25**	283.60	QWH-70-70-58	28	513.30	**30**	82.36
QWH-50-70-8	22	459.28	**29**	376.46	QWH-70-70-59	29	590.32	**30**	66.18
QWH-50-70-83	**28**	486.30	**28**	**175.30**	QWH-70-70-62	29	378.98	**30**	48.61
QWH-50-70-90	28	357.70	**30**	213.41	QWH-70-70-63	29	343.15	**30**	50.99
QWH-60-70-100	28	488.10	**30**	82.56	QWH-70-70-64	28	486.28	**30**	95.07
QWH-60-70-22	29	307.56	**30**	45.78	QWH-70-70-66	29	331.61	**30**	42.60
QWH-60-70-32	29	231.30	**30**	30.53	QWH-70-70-68	29	365.72	**30**	83.49
QWH-60-70-50	20	664.04	**30**	156.44	QWH-70-70-71	29	348.06	**30**	38.28
QWH-60-70-68	21	586.12	**30**	166.52	QWH-70-70-72	24	503.07	**30**	71.75
QWH-60-70-74	27	338.92	**30**	60.57	QWH-70-70-75	29	338.82	**30**	48.29
QWH-60-70-77	29	208.56	**30**	32.12	QWH-70-70-83	29	437.59	**30**	54.12
QWH-60-70-9	29	397.96	**30**	102.04	QWH-70-70-88	28	477.67	**30**	58.96
QWH-60-70-96	29	368.27	**30**	75.58	QWH-70-70-96	29	356.84	**30**	41.19
QWH-60-70-99	**30**	196.62	29	36.21					

Table 3: Detailed results of MMCOL and FastLSC in the random benchmark under ct =1000.

Results on COLOR03 Benchmark

Results on the COLOR03 benchmark are reported in Table 4. For all instances, FastLSC has the best performance in terms of success rate and run time, except slightly worse than MMCOL on only one instance. Specially, for 17 instances, these algorithms both can steadily find an arbitrary legal solution over 30 times, but FastLSC is about 2 to 3 times faster than MMCOL.

Figure 2 reports the average run time when FastLSC and MMCOL obtain the same success rate on all benchmarks, which indicates the effectiveness of the proposed FastLSC algorithm.

Analysis of Proposed Ideas

To study the effectiveness of our reduction rules, we compare RedAll with Red1 and Red23. red denotes the reduction number of vertices, while $\overline{red}$ denotes the average reduction number of vertices for each family. Table 5 shows that RedAll obtains better reduction ratio than Red1 and Red23, which illustrates the effectiveness of our reduction rules on all benchmarks. Moreover, for 65 instances, our FastLSP algorithm can obtain a legal solution by only using the reduction rules. We also add our reduction rules into MMCOL, resulting in a new algorithm MMCOL+RedAll. Results show that MMCOL+RedAll performs slightly better than MMCOL, but it is still worse than FastLSC.

To show the effectiveness of the secondary scoring function and our perturb function, we compare FastLSC with two alternative algorithms where FastLSC1 randomly puts 30% vertices into C simply by replacing lines 19 and 20 in Algorithm 3 for our perturb function and FastLSC2 uses the random selection method instead of our secondary scoring function. We ignore some instances where the difference of run time of the two algorithms is less than 0.1 seconds. The

Instance	MMCOL ct=1000s		FastLSC		MMCOL ct=100s		FastLSC	
	$\#suc$	$time$	$\#suc$	$time$	$\#suc$	$time$	$\#suc$	$time$
q*18*120	**30**	**<0.01**	**30**	**<0.01**	**30**	**<0.01**	**30**	**<0.01**
q*30*316	**30**	0.12	**30**	**0.05**	**30**	0.12	**30**	**0.05**
q*30*320	**30**	0.56	**30**	**0.13**	**30**	0.56	**30**	**0.13**
q*33*381*	**30**	164.7	**30**	**32.85**	13	41.46	**28**	27.87
q*35*405	**30**	17.07	**30**	**5.30**	**30**	17.07	**30**	**5.30**
q*40*528	**30**	12.52	**30**	**3.11**	**30**	12.52	**30**	**3.11**
q*5*10	**30**	**<0.01**	**30**	**<0.01**	**30**	**<0.01**	**30**	**<0.01**
q*50*750*	**1**	**332**	**1**	582.7	0	N/A	0	N/A
q*50*825*	**30**	85.76	**30**	**24.68**	20	61.29	**30**	24.68
q*60*1080*	0	N/A	**4**	384.8	0	N/A	0	N/A
q*60*1152*	29	347.4	**30**	47.30	1	26.08	**27**	36.66
q*60*1440	**30**	2.60	**30**	**1.17**	**30**	2.60	**30**	**1.17**
q*60*1620	**30**	0.84	**30**	**0.51**	**30**	0.84	**30**	**0.51**
q*70*2450	**30**	0.59	**30**	**0.44**	**30**	0.59	**30**	**0.44**
q*70*2940	**30**	0.54	**30**	**0.41**	**30**	0.54	**30**	**0.41**
qg.order100	**30**	424.7	**30**	**10.66**	0	N/A	**30**	10.66
qg.order30	**30**	0.05	**30**	**0.02**	**30**	0.05	**30**	**0.02**
qg.order40	**30**	0.19	**30**	**0.09**	**30**	0.19	**30**	**0.09**
qg.order60	**30**	1.78	**30**	**0.65**	**30**	1.78	**30**	**0.65**

Table 4: Results of MMCOL and FastLSC in the COLOR03 benchmark.

Instance	RedAll	Red1	Red23	Instance	RedAll	Red1	Red23
	red	red	red	Family	$\overline{red}$	$\overline{red}$	$\overline{red}$
qg.order100	0	0	0	50-30	0	0	0
qg.order30	0	0	0	50-40	0	0	0
qg.order40	0	0	0	50-50	0	0	0
qg.order60	0	0	0	50-60	**0.58**	0.19	0.39
q*18*120	**70**	26	66	50-70	**29.33**	9.19	19.57
q*30*316	**37**	8	24	50-80	**495.7**	495.24	495.53
q*30*320	**43**	23	24	60-30	0	0	0
q*33*381*	**14**	2	12	60-40	0	0	0
q*35*405	**41**	15	22	60-50	0	0	0
q*40*528	**28**	7	16	60-60	**0.12**	0.03	0.09
q*5*10.1	**10**	**10**	**10**	60-70	**14.18**	4.63	9.7
q*50*750*	**26**	2	21	60-80	**713.42**	516.63	713.39
q*50*825*	**2**	1	1	70-30	0	0	0
q*60*1080*	**8**	1	7	70-40	0	0	0
q*60*1152*	**9**	0	**9**	70-50	0	0	0
q*60*1440	0	0	0	70-60	**0.03**	**0.03**	0
q*60*1620	0	0	0	70-70	**6.58**	2.25	4.41
q*70*2450	0	0	0	70-80	**971.34**	162.24	958.53
q*70*2940	0	0	0				

Table 5: The information of our reduction rules. RedAll uses all reduction rules, Red1 only uses the first reduction rule and Red23 uses the latter two reduction rules. We use n-r to denote QWH-n-r.

results in Table 6 intuitively shows that the proposed two strategies play a key role in the FastLSC algorithm. Additional, combining the results in Tables 2, 4, and 6, it shows that FastLSC1 outperforms MMCOL, which can indirectly verify the effectiveness of our perturb function.

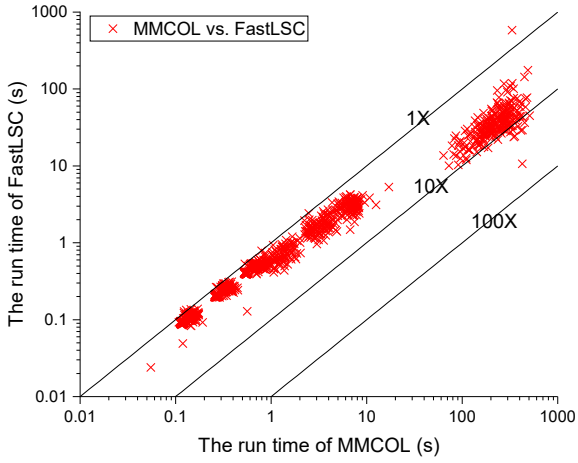

Figure 2: Average run time of MMCOL and FastLSC.

Instance	FastLSC1		FastLSC2		FastLSC	
Family	$\#suc$	$\overline{time}$	$\#suc$	$\overline{time}$	$\#suc$	$\overline{time}$
QWH-50-60	3000	1.22	3000	0.85	3000	**0.72**
QWH-50-70	**2925**	90.37	2915	91.37	2919	**70.63**
QWH-60-60	3000	5.12	3000	1.80	3000	**1.69**
QWH-60-70	1049	405.83	2999	67.52	2999	**40.99**
QWH-70-70	3000	**40.14**	3000	54.93	3000	45.39
Instance	$\#suc$	$time$	$\#suc$	$time$	$\#suc$	$time$
q*33*381*	30	37.17	30	33.56	30	**32.85**
q*35*405	30	5.4	30	5.96	30	**5.3**
q*40*528	30	3.32	30	3.61	30	**3.11**
q*50*750*	**1**	646.43	0	N/A	**1**	**582.66**
q*50*825*	29	79.69	30	**20.74**	30	24.68
q*60*1080*	0	N/A	4	717.62	4	**384.78**
q*60*1152*	3	543.94	30	52.29	30	**47.30**
q*60*1440	30	2.43	30	**0.97**	30	1.17
q*60*1620	30	0.6	30	**0.34**	30	0.51
qg.order100	30	11.35	30	16.47	30	**10.66**
qg.order60	30	0.67	30	1.19	30	**0.65**

Table 6: Comparing FastLSC with FastLSC1 and FastLSC2 on all benchmarks.

Conclusion

In this work, we propose several reduction rules, a conflict value selection heuristic and a novel perturbation mechanism for the LSC problem. Based on the above strategies, we develop a local search algorithm FastLSC. Results present that FastLSC outperforms the state-of-the-art heuristic algorithms.

In the future, we plan to further study variants of secondary scoring function (Li et al. 2020; Cai and Zhang 2021) in the context of the LSC problem to improve the algorithms. Also we would like to apply the novel perturbation mechanism into solving some other NP-hard problems.

Acknowledgements

This work was supported by NSFC (61806050, 61972063, 61976050), the Fundamental Research Funds for the Central Universities (2412020FZ030, 2412018QD022), QT202005.

References

Ansótegui, C.; del Val, A.; Dotú, I.; Fernández, C.; and Manya, F. 2004. Modeling choices in quasigroup completion: SAT vs. CSP. In *AAAI*, 137–142.

Barry, R. A.; and Humblet, P. A. 1993. Latin routers, design and implementation. *Journal of Lightwave Technology*, 11(5/6): 891–899.

Biro, M.; Hujter, M.; and Tuza, Z. 1992. Precoloring extension. I. Interval graphs. *Discrete Mathematics*, 100(1-3): 267–279.

Cai, S.; Luo, C.; and Zhang, H. 2017. From Decimation to Local Search and Back: A New Approach to MaxSAT. In *IJCAI*, 571–577.

Cai, S.; and Zhang, X. 2021. Deep Cooperation of CDCL and Local Search for SAT. In *SAT*, 64–81.

Colbourn, C. J. 1984. The complexity of completing partial latin squares. *Discrete Applied Mathematics*, 8(1): 25–30.

Colbourn, C. J. 2010. *CRC handbook of combinatorial designs*. CRC press.

Colbourn, C. J.; Klove, T.; and Ling, A. C. 2004. Permutation arrays for powerline communication and mutually orthogonal latin squares. *IEEE Transactions on Information Theory*, 50(6): 1289–1291.

Glover, F.; and Laguna, M. 1998. Tabu search. In *Handbook of combinatorial optimization*, 2093–2229. Springer.

Gogate, V.; and Dechter, R. 2011. SampleSearch: Importance sampling in presence of determinism. *Artificial Intelligence*, 175(2): 694–729.

Gomes, C.; and Shmoys, D. 2002. Completing quasigroups or latin squares: A structured graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and Generalizations*, 22–39.

Gomes, C. P.; Regis, R. G.; and Shmoys, D. B. 2004. An improved approximation algorithm for the partial Latin square extension problem. *Operations Research Letters*, 32(5): 479–484.

Hajirasouliha, I.; Jowhari, H.; Kumar, R.; and Sundaram, R. 2007. On completing latin squares. In *Annual Symposium on Theoretical Aspects of Computer Science*, 524–535.

Haraguchi, K. 2015. An efficient local search for partial latin square extension problem. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, 182–198. Springer.

Haraguchi, K. 2016. Iterated local search with Trellis-neighborhood for the partial Latin square extension problem. *Journal of Heuristics*, 22(5): 727–757.

Jin, Y.; and Hao, J.-K. 2019. Solving the Latin square completion problem by memetic graph coloring. *IEEE Transactions on Evolutionary Computation*, 23(6): 1015–1028.

Kumar, S. R.; Russell, A.; and Sundaram, R. 1999. Approximating latin square extensions. *Algorithmica*, 24(2): 128–138.

Lakić, N. 2001. The application of Latin square in agronomic research. *Journal of Agricultural Sciences (Belgrade)*, 46(1): 71–77.

Laywine, C. F.; and Mullen, G. L. 1998. *Discrete mathematics using Latin squares*, volume 49. John Wiley & Sons.

Li, B.; Zhang, X.; Cai, S.; Lin, J.; Wang, Y.; and Blum, C. 2020. NuCDS: An Efficient Local Search Algorithm for Minimum Connected Dominating Set. In *IJCAI*, 1503–1510.

Wang, Y.; Cai, S.; Chen, J.; and Yin, M. 2020a. SCCWalk: An efficient local search algorithm and its improvements for maximum weight clique problem. *Artificial Intelligence*, 280: 103230.

Wang, Y.; Cai, S.; Pan, S.; Li, X.; and Yin, M. 2020b. Reduction and local search for weighted graph coloring problem. In *AAAI*, volume 34, 2433–2441.

Xu, Z.; He, K.; and Li, C.-M. 2019. An iterative Path-Breaking approach with mutation and restart strategies for the MAX-SAT problem. *Computers & Operations Research*, 104: 49–58.